REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** February 13, 2008	**3. REPORT TYPE AND DATES COVERED** Final, January 2006 - November 2008

4. TITLE AND SUBTITLE Final Report for Organization-based Model-driven Development of High-assurance Multiagent Systems	**5. FUNDING NUMBERS** FA9550-06-1-0058
6. AUTHOR(S) Scott A. DeLoach and Robby	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kansas State University Department of Computing & Information Sciences 234 Nichols Hall Manhattan, KS 66506-2302	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM (David Luginbuhl) Suite 325, Room 3112 875 Randolph Street Arlington VA 22203-1768	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-OSR-VA-TR-2012-0667

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT A- Approve for public Release	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

This report presents the final results the research grant "Organization-based Model-driven Development of High-assurance Multiagent Systems" performed by Dr. Scott A. DeLoach and Dr. Robby at Kansas State University. The goal of this research is to develop methods, techniques, and tools to allow developers to design and build highly adaptive distributed systems that are assured of meeting specific design goals. Specifically, there were three key focus areas in this research: (1) to develop a model-driven software engineering methodology for the development of high-assurance, highly adaptive multiagent systems, (2) to explore policy-based mechanisms for specifying application-specific properties and metrics for adaptive multiagent systems, and (3) to develop an integrated set of tools to support our proposed software methodology, including automated verification capabilities based on lightweight and model checking approaches. This report details the key results in each of these areas.

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 98
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT**

Scott A. DeLoach & Robby

Kansas State University
234 Nichols Hall
Manhattan, KS 66506-2302

Phone: (785) 532-6350
Fax: (785) 532-7353
E-mail: sdcloach@cis.ksu.edu
robby@ksu.edu

Final Report

Organization-based Model-driven Development of High-assurance Multiagent Systems

Grant Number: FA9550-06-1-0058

FEBRUARY 27, 2009

20120918109

1 Introduction

This report presents the final results the research grant "Organization-based Model-driven Development of High-assurance Multiagent Systems" performed at Kansas State University. The goal of this research is to develop methods, techniques, and tools to allow developers to design and build highly adaptive distributed systems that are assured of meeting specific design goals. Specifically, there were three key focus areas in this research:

(1) to develop a model-driven software engineering methodology for the development of high-assurance, highly adaptive multiagent systems,

(2) to explore policy-based mechanisms for specifying application-specific properties and metrics for adaptive multiagent systems, and

(3) to develop an integrated set of tools to support our proposed software methodology, including automated verification capabilities based on lightweight and model checking approaches.

1.1 Key Results

During the project, we had several key results. These are outlined here and discussed in more detail below.

We develop the Organization-based Multiagent Systems Engineering (O-MaSE) methodology, which is a customizable, model-driven software engineering methodology for the development of high-assurance, highly adaptive multiagent systems. The O-MaSE Process Framework is a methodology framework that allows process engineers to construct custom agent-oriented processes using a set of method fragments, all of which are based on a common metamodel. To achieve this, we define O-MaSE in terms of a metamodel, a set of method fragments, and a set of guidelines. The O-MaSE *metamodel* defines a set of analysis, design, and implementation concepts and a set of constraints between them. The *method fragments* define how a set of analysis and design products may be created and used within O-MaSE. Finally, *guidelines* define how the method fragment may be combined to create valid O-MaSE processes, which we refer to as O-MaSE compliant processes.

We defined and exploited a new type of policy for use in specifying application-specific properties for adaptive multiagent systems, called guidance policies. We developed a *formal* trace-based foundation for *law* and *guidance* policies and a conflict resolution strategy for choosing between which guidance policies to violate under circumstances where not all guidance policies can be followed. Next, we developed an organizational policy-based approach to self-tuning and a generalized algorithm based on Q-Learning to implement the policy-based tuning. Finally, we developed a formal method for generating specifications multiagent systems from abstract qualities and a method of automatically discovering potential conflicts in abstract qualities given a system design. The approach uses highly-optimized model checking techniques to generate system traces from which policies are mined.

We created a modeling checking based framework for producing predictive metrics for adaptive, complex systems. Here we combined the relationships between the goals, roles, and agents in such systems to enumerate all legal execution traces and then provide metrics based on those traces. Specifically, we defined a flexibility metric that is capable of predicting how many different ways a system may achieve the overall system goals as well as an *occurrence* metrics that measures how many times a particular goal, role, or agent is actually used in achieving the overall system goal. Both these sets of goals are computed at design time, thus allowing the designer to make requisite changes much cheaper than waiting until simulation or system testing.

Develap the agentTaal III taalset as a way ta integrate the research dane in this praject, as well as far demanstrating the effectiveness and relevance af aur appraaches. The agentTool III (aT^3) development environment provides traditional and advanced model creation tools to support the analysis, design, and implementation of multiagent systems. aT^3 also provides a lightweight verification component and a Bogor-based metrics computation component. In addition, aT^3 provides the ability to compose, verify, and maintain customized O-MaSE complaint processes. In all, aT^3 has five components: the graphical editor, the process editor, the verification framework, the metrics computation component, and the code generation facility.

1.2 Publications

The following publications were a result of the work done on this grant, either entirely or in part. The list includes three journal articles, one book chapter, and eight conference or workshop proceedings papers.

1. Scott DeLoach, Lin Padgham, Anna Perini, Angelo Susi, and John Thangarajah. Using Three AOSE Toolkits to Develop a Sample Design. International Journal of Agent Oriented Software Engineering. (in press).
2. Scott A. DeLoach. Moving Multiagent Systems from Research to Practice, in Special section on Future of software engineering and multi-agent systems, International Journal of Agent-Oriented Software Engineering (IJAOSE). (in press).
3. Scott A. DeLoach. Organizational Model for Adaptive Complex Systems. in Virginia Dignum (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global: Hershey, PA. ISBN: 1-60566-256-9 (March 2009).
4. Juan C. Garcia-Ojeda, Scott A. DeLoach, and Robby. agentTool Process Editor: Supporting the Design of Tailored Agent-based Processes. Proceedings of the 24th Annual ACM Symposium on Applied Computing, Honolulu, Hawaii, USA. March 8 - 12, 2009.
5. Scott Harmon, Scott DeLoach, and Robby. From Abstract Qualities to Concrete Specification using Guidance Policies. Proceedings of the Proceedings of 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Decker, Sichman, Sierra, and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary.
6. Scott A. DeLoach, Walamitien Oyenan and Eric T. Matson. A Capabilities Based Model for Artificial Organizations. Journal of Autonomous Agents and Multiagent Systems. Volume 16, no. 1, February 2008, pp. 13-56.
7. Scott J. Harmon, Scott A. DeLoach, Robby, and Doina Caragea. Leveraging Organizational Guidance Policies with Learning to Self-Tune Multiagent Systems Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Isola di San Servolo (Venice), Italy, October 20-24, 2008.
8. Lin Padgham, Michael Winikoff, Scott DeLoach, and Massimo Cossentino. A Unified Graphical Notation for AOSE. Proceedings of the 9th International Workshop on Agent Oriented Software Engineering, Estoril Portugal, May 2008.
9. Scott A. DeLoach. Developing a Multiagent Conference Management System Using the O-MaSE Process Framework. In Michael Luck (eds.), Agent-Oriented Software Engineering VIII: The 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007), LNCS 4951, 171-185, Springer-Verlag: Berlin.

10. Juan C. Garcia-Ojeda, Scott A. DeLoach, Robby, Walamitien H. Oyenan and Jorge Valenzuela. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. In Michael Luck (eds.), Agent-Oriented Software Engineering VIII: The 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007), LNCS 4951, 1-15, Springer-Verlag: Berlin.
11. Scott Harmon, Scott A. DeLoach, and Robby. Trace-based Specification of Law and Guidance Policies for Multiagent Systems. The Eighth Annual International Workshop "Engineering Societies in the Agents World" (ESAW 07) Athens, Greece, October, 2007.
12. Robby, Scott A. DeLoach, Valeriy A. Kolesnikov. Using Design Metrics for Predicting System Flexibility. in Luciano Baresi, Reiko Heckel (eds.) Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Springer LNCS Vol 3922, pp 184-198, 2006.

The report is laid out as follows. In the next subsections, we describe general background material relevant to each our focus area. First, we describe the Organizational Model for Adaptive Complex Systems (OMACS) followed by a description of the customizations to the model checking techniques and Bogor engine developed in this research, and the concept of system traces used extensively in the policy and metrics focus areas. We then present the three focus areas Policies for Multiagent Systems, Organization-Based Multiagent Systems Engineering and the agentTool III Development Environment, and Predictive Metrics for Multiagent Systems.

1.3 Background

1.3.1 OMACS

The Organizational Model for Adaptive Complex Systems (OMACS) framework (DeLoach, Oyenan, & Matson, 2008) defines an organizational structure that allows multiagent teams to reconfigure at runtime, thus enabling them to cope with unpredictable situations in a dynamic environment. This framework provides all the knowledge required in order for a multiagent system to know itself and be able to reason about its own status. Hence, multiagent teams are not limited by a predefined set of configurations, and they can have the appropriate information about their team, thus enabling them to reconfigure in order to achieve their team goals more efficiently and effectively. The designer provides high-level guidance about the organization (team of agents), which will enable it to self-configure based on the current goals and team capabilities. These characteristics make the OMACS model very suitable for designing autonomic multiagent systems.

1.3.1.1 The OMACS metamodel

The OMACS defines an organization as: (1) a set of goals (G) that the team is attempting to accomplish, (2) a set of roles (R) that must be played to achieve those goals, (3) a set of capabilities (C) required to play those roles, and (4) a set of agent (A) assigned to roles in order to achieve the organization goals; there are more entities defined in OMACS which are however not relevant for this paper. Figure 1 shows the OMACS metamodel using the standard UML notation. Only the entities discussed in this report are shown. The reader is referred to (DeLoach, Oyenan, & Matson, 2008) for the complete model.

1.3.1.2 Goals

Goals are a high-level description of what the system is supposed to be doing (Russel & Norvig, 20002). In OMACS, goals are represented by the Goal Model for Dynamic Systems (GMoDS), which includes the goal definitions, goal decomposition, and the relationship between the goals and their subgoals; subgoals are either conjunctive (AND-decomposed) or disjunctive (OR-decomposed) (van Lamsweerde, Darimontm, & Letier, 1998). Typically, each organization has a top-level goal that is decomposed into refined subgoals. Eventually, this top-level goal is refined into a set of leaf goals that will be actually pursued by the organization. The set of all organizational goals is denoted as *G*. The active goal set, *Ga* (where $Ga \subset G$), is the set of goals that an organization is trying to achieve at the current time. *Ga* changes as new goals are inserted or current goals are achieved. Note that only leafs goals can be in *Ga*.

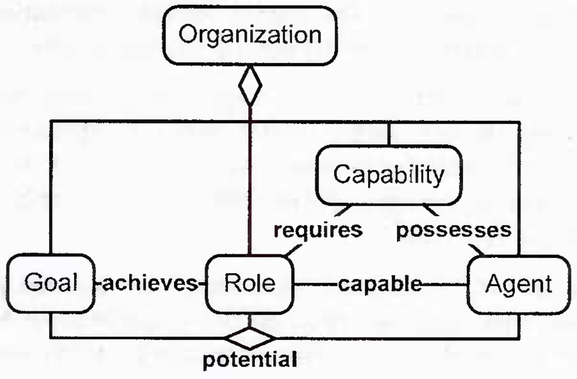


Figure 1. Simplified OMACS Metamodel

GMoDS takes the initial Goal Model and adds additional information to capture the dynamism associated with the system. GMoDS introduces three concepts into AND/OR goal modeling approaches to handle goal sequencing, the creation of goal instances, and parameterized goals. To capture the sequencing of OMACS-based systems, there is a time-based relationship that exists between goals. We say goal *g1 precedes* goal *g2*, if *g1* must be satisfied before any part of *g2* can be satisfied. This allows the organization to work on one part of the goal tree at a time. During the pursuit of specific goals, events may occur that cause the instantiation of new goals. If an event can occur during the pursuit of goal *g1* that causes the instantiation of goal *g2*, we say that *g1 triggers g2*. Goals without a specific trigger are created at system initialization, while other goals are created when specific events occur. These instantiated goals may be parameterized to allow the goal to take on a context sensitive meaning. For instance, in a conference management system, we might have a goal to review a paper. However, this goal is ambiguous until we specify which specific paper is to be reviewed. Thus, we add a parameter to the goal to specify the paper to be reviewed.

1.3.1.3 Roles

Roles are a high level description of how to achieve some particular goals (Ferber, Gutknecht, Jonker, Müller, & Treur, 2002). In OMACS, each organization has a set of roles that it can use to achieve its goals. The *achieves* function, which associates a score between 0 and 1 to each <goal, role> pair, tells how well that particular role can be used to achieve that goal (1 being the maximum score). In addition, each role requires a set of capabilities, which are inherent to particular agents. Agents must possess all the required capabilities in order to be considered as a potential candidate to assume that role.

1.3.1.4 Capabilities

In the OMACS framework, capabilities are fundamental in determining which agents can be assigned to what roles in the organization (Matson &DeLoach, 2003). Agent may possess two types of capabilities: (1) hardware capabilities like actuator or effectors, and (2) software capabilities like computational algorithms or resources usage.

1.3.1.5 Agents

Agents represent the autonomics elements of the system (Kephart & Chess, 2003). Within an OMACS organization, agents have the ability to communicate with each other, accept assignments to play roles that match their capabilities, and work to achieve their assigned goals. Each agent is responsible for managing its own state and its interactions with the environment and with other agents. Once the system provides goals in the form of high-level notions, the agents are expected to determine themselves what behavior is necessary to fulfill them. When assuming a role, the agent needs to follow the guidance provided by the high level description supplied by that role. For that, a plan on how to play a role needs to be provided at design time either by the role or the agent designer. In OMACS, a tuple <*a*,*r*,*g*> represents an assignment if agent *a* has been assigned by the organization to play role *r* in order to achieve goal *g*. As discussed above, however, only agents with the right set of capabilities may be assigned to a role. The assignment set Φ represents the set of all the current assignments in the organization. To capture a given agent's capabilities, OMACS defines a *possesses* function, which maps each <agent, capability> pair to a value between 0 and 1, describing the quality of the capability possessed by an agent (1 representing the maximum quality).

1.3.2 The Bogor Model Checking Framework

Bogor (Robby, Dwyer, & Hatcliff, 2003) is an extensible and highly modular model checking framework that enables effective incorporation of domain knowledge into verification models, associated model checking algorithms, and optimizations, by focusing on the following principles:

- **Direct Support of High-level Language Features:** Bogor provides a rich base modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. For these primitives, we have extended Bogor's default algorithms to support state-of-the-art model reduction/optimization techniques that we have developed by customizing for modern software existing techniques such as collapse compression, heap symmetry, thread symmetry, and partial-order reductions.
- **Extensible Modeling Language:** Bogor's modeling language can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g., multi-agent systems, avionics, security protocols, etc.) and a particular level of abstraction (e.g., design models, source code, byte code, etc.)
- **Open Modular Architecture:** Bogor's well-organized module facility allows new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.) to be easily swapped in to replace Bogor's default model checking algorithms.
- **Design for Encapsulation:** Bogor is written in Java and comes wrapped as a plug-in for Eclipse, an open source and extensible universal tool platform originally from IBM. This allows Bogor to be deployed as a stand-alone tool with a rich graphical user interface and a variety of visualization facilities, or encapsulated within other development or verification tools for a specific domain.

In short, Bogor aims to be not only a robust and feature-rich software model checking tool that handles the language constructs found in modern large-scale software system designs and implementations, it also aims to be a model checking *framework* that enables researchers and engineers to rapidly create families of domain-specific model checking engines.

1.3.3 Multiagent Traces

There are several observable events in an OMACS system. A *system event* is simply an action taken by the system. In this paper, we are concerned with specific actions that the organization takes. For instance, an assignment of an agent to a role is a system event. The completion of a goal is also a system event. In an OMACS system, we can have the system events of interest shown in Table 1.

At any stage in a multiagent system, there may be certain properties of interest. Some may be domain-specific (only relevant to the current system), while others may be general properties such as the number of roles an agent is currently playing. State properties that are relevant to the examples we are presenting in the next section are shown in Table 1.

Table 1. Events and Properties of Interest

System Events	**Definition**
$C(G_i)$	goal g_i has been completed.
$T(G_i)$	goal g_i has been triggered.
$A(a_i, r_j, g_k)$	agent a_i has been assigned role r_j to achieve goal g_k.
Properties	**Definition**
a.reviews	the number of reviews agent a has performed.
a.vacuumedRooms	the number of rooms agent a has vacuumed.

1.3.3.1 System Traces

In order to describe multiagent system execution, we use the notion of a system trace. An (abstract) *system trace* is a projection of system execution with only desired state and event information preserved (role assignments, goal completions, domain-specific state property changes, etc). In our approach, we are only concerned with the events and properties given above and only traces that result in a successful completion of the system goal. Let *E* be an event of interest and *P* be a property of interest. A *change of interest* in a property is a change for which a system designer has made some policy. For example, if a certain integer should never exceed 5, a change of interest would be when that integer became greater than 5 and when that integer became less than 5. Thus a change of interest in a property is simply an abstraction of all the changes in the property. ΔP indicates a change of interest in property *P*. A system trace may contain both events and changes of interest in properties. Changes of interest in properties may be viewed as events, however, for simplicity we include both and use both interchangeably. Thus, a system trace is defined as:

$$E_1 \rightarrow E_2 \rightarrow \ldots \qquad (1)$$

As shown in Equation 1, a trace is simply a sequence of events. An example sub-trace of a multiagent system, where g_1 is a goal, a_1 is an agent, and r_1 is a role, might be:

$$\ldots\ T(g_1) \rightarrow A(a_1,r_1,g_1) \rightarrow C(g_1)\ \ldots \qquad (2)$$

Equation 2 means that goal g_1 is triggered, then agent a_1 is assigned role r_1 to achieve goal g_1, finally, goal g_1 is completed.

We use the terms *legal trace* and *illegal trace*. An *illegal trace* is an execution we do not want our system to exhibit, while a *legal trace* is an execution that our system may exhibit. Intuitively, policies cause some traces to become *illegal*, while others remain *legal*.

We are able to use the notion of system traces because the framework we are using to build multiagent systems constructs mathematically specified models (e.g., DeLoach et al., 2008, Miller, 2007) of various aspects of the system (e.g., goal model, role model, etc.). This can be leveraged to formally specify policies as restrictions of system traces. Once we have a formal definition of system traces, we can leverage existing research on property specification and concurrent program analysis.

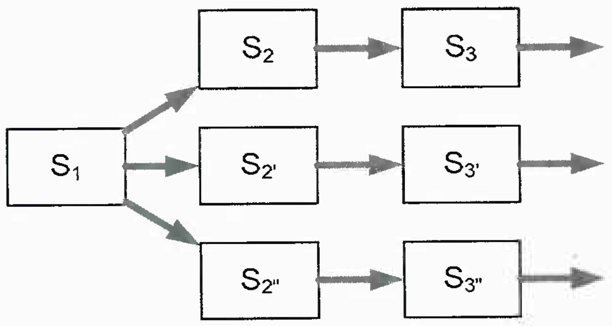


Figure 2. System Traces

Figure 2 shows a graphical depiction of system traces showing the fan-out at a decision point. Starting at state S_1, the system has three options, after that decision is made, and the traces diverge. The set of traces form a computation tree. We use this structure to determine what choices to make at each decision point in order to maximize (or minimize) some metric. The choice may be what role to or what agent to use to achieve a goal, or even what goal to pursue (in the case of OR goals). It is important to note that the divergence in the traces may also happen by changes in goal parameters, which are normally beyond the control of the system. Thus, the metrics and policies generated may need to take into account some aspect of the goal parameter.

We used Bogor (Robby, Dwyer, & Hatcliff, 2003) to generate the system traces using the models defined by the OMACS meta-model. Bogor is an extensible, state of the art model checker. Our customized Bogor uses an extended GMoDS goal model, a role model, and an agent model to generate traces of a multiagent system. We have extended the GMoDS model with user-adjustable maximum and minimum bounds on the *triggers* relationship in order to limit the exploration of the trace-space in the model checker. The traces consist of a sequence of *agent goal assignment* achievements. We generate only traces in which the top-level goal is achieved (system success).

Agent goal assignment: An agent goal assignment is defined as a tuple, $\langle G(x), R, A \rangle$, containing a parameterized goal $G(x)$, a role R, and an agent A. The goal's parameter x may be any domain specific specialization of the goal G given at the time the goal is dispatched (triggered) to the system.

2 Policies for Multiagent Systems

As computer systems have been charged with solving problems of greater complexity, the need for distributed, intelligent systems have increased. As a result, there has been a focus on creating systems based on interacting autonomous agents. This investigation has created an interest in multiagent systems and multiagent system engineering, which proscribes formalisms and methods to help software engineers design multiagent systems. One aspect of multiagent systems that is receiving considerable attention is the area of policies. These policies have been used to describe the properties of a multiagent system–whether that be behavior or some other design constraints. Policies are essential in designing societies of agents that are both predictable and reliable (Kephart & Chess, 2003). Policies have traditionally been interpreted as properties that must always hold. However, this does not capture the notion of policies in human organizations, as they are often used as normative *guidance*, not strict *laws*. Typically, when a policy cannot be followed in a multiagent system, the system cannot achieve its goals, and thus, it cannot continue to perform. In contrast, policies in human organizations are often suspended in order to achieve the overall goals of the organization. We believe that such an approach could be extremely beneficial to multiagent systems residing in a dynamic environment. Thus, we want to enable developers to guide the system without constraining it to the point where it cannot function effectively or loses its autonomy.

A robust system should adapt to environments, recover from failure, and improve over time. In human organizations, policies evolve over time and are adapted to overcome failures. New policies are introduced to avoid unfavorable situations. A robust organization-based multiagent system should also be able to evolve its policies and introduce new ones to avoid undesirable situations.

Organization-based multiagent systems engineering has been proposed as a way to design complex systems that adapt to their environment (Bernon, Gleizes, & Picard, 2005, DeLoach et al., 2007). Agents interact and can be given tasks depending on their individual capabilities. The system designer, however, may not have planned for every possible environment within which the system may be deployed. The agents themselves may exhibit particular properties that were not initially anticipated. As multiagent systems grow, configuration and tuning of these systems can become as complex as the problems they claim to solve.

In this section, we introduce guidance policies as formal rules that are applied to OMACS systems. This is consistent with usage in formalizations such as KAoS (Uszok et al., 2003) and PONDER (Damianou, Dulay, Lupu, & Sloman, 2000). Guidance policies are a trace-based formalization of policies 'that need not always be followed'. They must be followed when the system can still progress toward achieving its goal. If the system cannot continue with the guidance polices, they may be suspended temporarily. The guidance policies may be arranged in a *more-important-than* relation, creating a set of lattices. The policies are suspended from least-important to most-important. Thus it is possible to have conflicting guidance policies and yet still have a valid and viable system.

2.1 Contributions

Our work on multiagent policies was divided into three phases. First, we defined the notion of guidance policies as a way to provide guidance to the system while still maintaining flexibility. Second, we developed an approach to create a self-tuning mechanism based on the use of guidance policies. Finally, we applied guidance policies to the problem of satisfying abstract qualities/non-functional requirements in an adaptive system. We validated each all of the work in this area using simulation and experimentation. The main contributions of each of the three phases are described below.

1) We developed (1) a *formal* trace-based foundation for *law* and *guidance* policies, (2) a conflict resolution strategy for choosing between which guidance policies to violate.
2) We developed (1) an organizational policy-based approach to self-tuning, (2) a generalized algorithm based on Q-Learning to implement the policy-based tuning.
3) We developed (1) a formal method for generating specifications at design time for multiagent systems from abstract qualities, (2) a method of automatically discovering potential conflicts in abstract qualities given a system design.

2.2 Background

Policies have been considered for multiagent systems for some time. Efforts have been made to characterize, represent, and reason (Bradshaw et al., 2003) about policies in the context of multiagent systems. Policies have been referred to as laws in the past. Yoav Shoham and Moshe Tennenholtz wrote in (Shoham & Tennenholtz, 1995) about *social laws* for multiagent systems. They showed how policies could help a system to work together, similar to how our rules of driving on a predetermined side of the road help the traffic to move smoothly. There has also been work on detecting global properties (Stoller, Unnikrishnan, & Liu, 2000) of a distributed system, which could in turn be used to suggest policies for that system. Policies have also been proposed as a way to help assure that agents and that the entire multiagent system behave within certain boundaries. They have also been proposed as a way to specify security constraints in multiagent systems (Kagal, Finin, & Joshi, 2003, Paruchuri, Tambe, Ordóñez, & Kraus, 2006). There has been work to define policy languages by defining a description logic (Uszok et al., 2003). Policies have also been referred to as *norms*. Much work has been done on the formal specification of these norms (Artikis, Sergot, & Pitt, 2007). We are taking this formal approach in our specification of guidance and law policies. Norms, however, are usually associated with *open systems*–while we are concerned with *closed, cooperative systems*. We want to use formal methods to prove whether a given system will abide by the policies as expected. Thus, we must give our guidance policies for multiagent societies a solid formal foundation. In order to achieve this end, we borrow concepts that are widely used in program analysis, in particular, model checking. Taking a model checking approach to policies has been done (Viganò & Colombetti, 2007) and is a natural extension of program analysis.

2.3 Exemplar Systems

We used several example systems throughout this research project. Two of the more complex systems are explained below. Two others are introduced where needed due to their simplicity and limited scope.

2.3.1 Conference Management Example

A well-known example in multiagent systems is the Conference Management (Zambonelli, Jennings, & Wooldridge, 2001, DeLoach, 2002) example. The Conference Management example models the workings of a scientific conference, for example, authors submit papers, reviewers review the submitted papers, and certain papers are selected for the conference and printed in the proceedings. Figure 3 shows the complete goal model for the conference management example, which we are using to illustrate our policies. In this example, a multiagent system represents the goals and tasks of a generic conference paper management system. Goals of the system are identified and are decomposed into subgoals.

The top-level goal, *0. Manage conference submissions*, is decomposed into several "*and*" subgoals, which means that in order to achieve the top goal, the system must achieve all of its subgoals. These subgoals are then associated through precedence and trigger relations. The *precedes* arrow between goals indicates that the source of the arrow must be *achieved* before the destination can become active. The

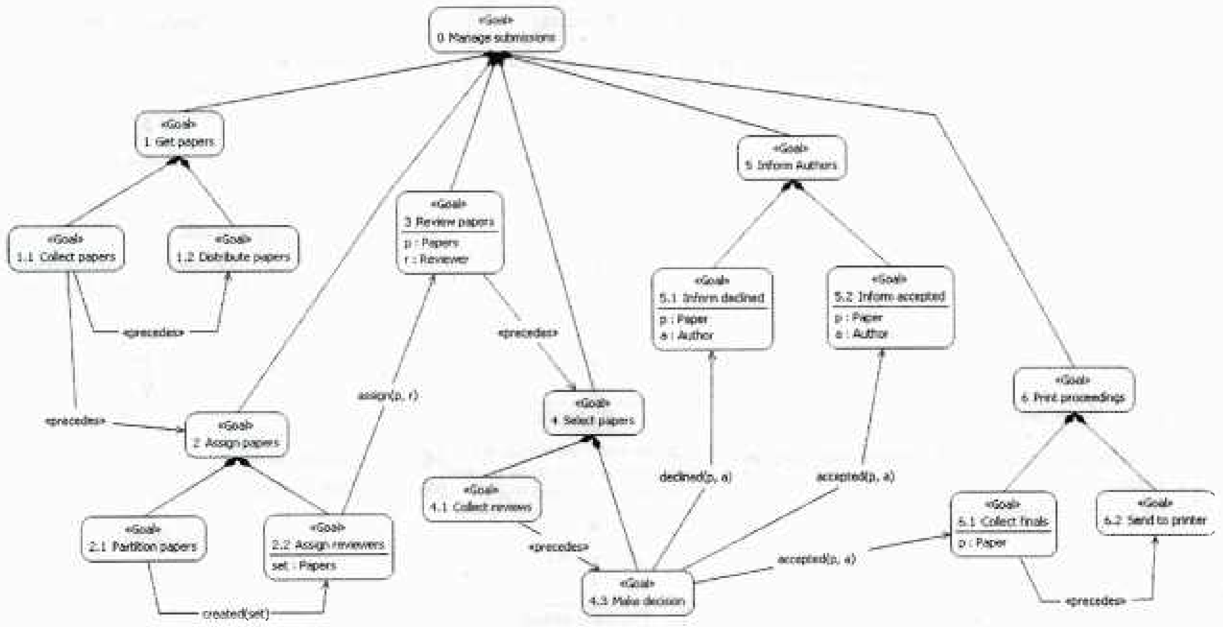


Figure 3. Conference Management System Goal Model

triggers arrow indicates that the domain-specific event in the source may trigger the goal in the destination. The *occurs* arrow from a goal to a domain-specific event indicates that while pursuing that goal, said event may occur. A goal that triggers another goal may trigger multiple instances of that goal.

Leaf goals are goals that have no children. The leaf goals in this example consist of Collect papers, Distribute papers, Partition papers, Assign reviewers, Collect reviews, Make decision, Inform accepted, Inform declined, Collect finals, and Send to printer. For each of these leaf goals to be achieved, agents must play specific roles. The roles required to achieve the leaf goals are depicted in Figure 4. The role model gives seven roles as well as two outside actors. Each role contains a list of leaf goals that the role can achieve. For example, the Assigner role can achieve the Assign reviewers leaf goal. In GMoDS, roles only achieve leaf goals. The arrows between the roles indicate interaction between particular roles. For example, once the agent playing the Partitioner role has some partitions, it will need to hand off these partitions to the agent playing the Assigner role. OMACS allows an agent to play multiple roles simultaneously, as long as it has the capabilities required by the roles and it is allowed by the policies.

2.3.2 Robotic Floor Cleaning Example

Another example to illustrate the usefulness of the concept of guidance policies is the Cooperative Robotic Floor Cleaning Company Example (CRFCC), which was first presented by Robby et al. in (Robby, DeLoach, & Kolesnikov, 2006). In this example, a team of robotic agents cleans the floors of a building. The team has a map of the building as well as indications of whether a floor is tile or carpet. Each team member will have a certain set of capabilities (e.g. vacuum, mop, etc). These capabilities may become defective over time. In their analysis, Robby et al. showed how breaking up the capabilities affected a team's flexibility to overcome loss of capabilities. We have extended this example by giving the information that the vacuum cleaner's bag needs to be changed after vacuuming three rooms. Thus, we want to minimize the number of bag changes. For this, we introduce a guidance policy and show how it affects the performance of the organization.

The goal model for the CRFCC system is fairly simple. As seen in Figure 5, the overall goal of the system (Goal 0) is to clean the floors. This goal is decomposed into three conjunctive subgoals: *1. Divide Area*, *2.*

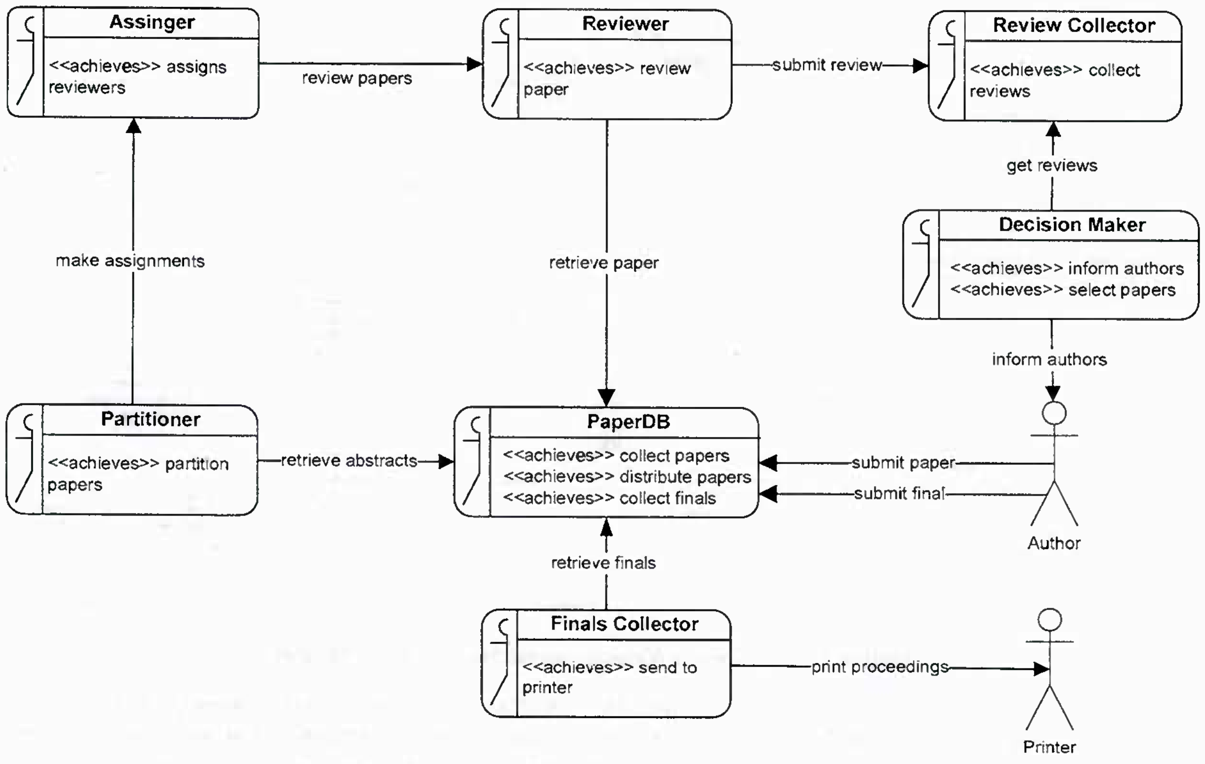

Assinger
<<achieves>> assigns reviewers
review papers
Reviewer
<<achieves>> review paper
submit review
Review Collector
<<achieves>> collect reviews
get reviews
Decision Maker
<<achieves>> inform authors
<<achieves>> select papers
make assignments
retrieve paper
inform authors
Partitioner
<<achieves>> partition papers
retrieve abstracts
PaperDB
<<achieves>> collect papers
<<achieves>> distribute papers
<<achieves>> collect finals
submit paper
submit final
Author
retrieve finals
Finals Collector
<<achieves>> send to printer
print proceedings
Printer

Figure 4. Conference Management Role Model

Pickup, and *3. Clean*. The *3. Clean* goal is decomposed into two disjunctive goals: *3.1 Sweep & Mop* and *3.2 Vacuum*. Depending on the floor type, only one subgoal must be achieved to accomplish the *3. Clean* goal. If an area needs to be swept and mopped (i.e. it is tile), then goal *3.1 Sweep & Mop* is decomposed into two conjunctive goals: *3.1.1 Sweep* and *3.1.2 Mop*. After an agent achieves the *1. Divide area* goal, a certain number of *2. Pickup* goals will become active (depending on how many pieces the area is divided into). After the *2. Pickup* goals are completed, a certain number of *3. Clean* goals become active, again depending on how many pieces the area was broken into. This then will activate goals for the tile areas (*3.1.1 Sweep* and *3.1.2 Mop*) as well as goals for the carpeted areas (*3.2 Vacuum*).

Figure 4 gives the role model for the CRFCC. In this role model, each leaf goal of the system is achieved by a specific role. The role model may be designed many different ways depending on the system's goal, agent, and capability models. Thus, depending on the agents and capabilities available, the system designer may choose different role models. For this paper, we will look at just one of these possible role models. For example, the *Pickuper* role requires the *search*, *move* and *pickup* capabilities. Thus, in order to play this role, an agent must possess all three capabilities.

2.4 Guidance Policies

Policies may restrict or proscribe behaviors of a system. Policies concerning agent assignments to roles have the effect of constraining the set of possible assignments. This can greatly reduce the search space when looking for the optimal assignment set (Zhong & DeLoach, 2006).

Other policies can be used for verifying that a goal model meets certain criteria. This allows the system designer to state more easily the properties of the goal model that may be verified against candidate

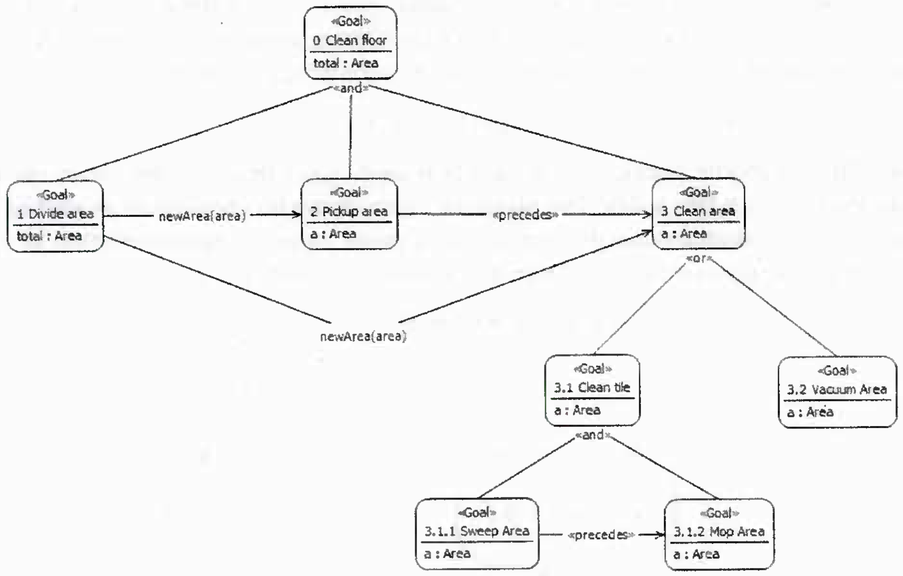


Figure 5 . CRFCC Goal Model

goal models at design time. For example, one might want to ensure that our goal model in Figure 3 will always trigger a *Review Paper* goal for each paper submitted.

Table 2. Conference Management Role Model

Role Name	Req. Capabilities	Goals Achieved
Organizer	org	Divide Area
Pickuper	search, move, pickup	Pickup area
Sweeper	move, sweep	Sweep area
Mopper	move, mop	Mop area
Vacuummer	move, vacuum	Vacuum area

Yet, other policies may restrict the way that roles can be played. For example, *when an agent is moving down the sidewalk it always keeps to the right*. These behavior policies also restrict how an agent interacts with its environment, which in turn means that they can restrict protocols and agent interactions. One such policy might be that an agent playing the *Reviewer* role must always give each review a unique number. These sort of policies rely heavily on domain-specific information. Thus it is important to have an ontology for relevant state and event information prior to designing policies (DiLeo, Jacobs, & DeLoach, 2002).

2.4.1 Language for policy analysis

To describe our policies, we use temporal equations with quantification similar to (Corbett, Dwyer, Hatcliff, & Robby, 2002). This may be converted into Linear Temporal Logic (LTL) (Manna & Pnueli, 1991) or Büchi automata (Büchi, 1960) for infinite system traces, or to something like Quantified Regular Expressions (Olender & Osterweil, 1990) for finite system traces. The equations consist of predicates over goals, roles, events, and assignments (recall that an assignment is the joining of an agent and role for the purpose of achieving a goal). The temporal operators we currently use are as follows: $\Box(x)$, meaning x holds always; $\Diamond(x)$, meaning x holds eventually; and $x\ \mathcal{U}\ y$, meaning x holds until y holds.[1] We use a mixture of state properties as well as events (Chaki, Clarke, Ouaknine, Sharygina, & Sinha, 2004) to obtain compact and readable policies. An example of one such policy equation is:

$$\forall a_1\text{: Agents, } \mathcal{L}: \square(sizeOf(a_1.reviews) \leq 5) \qquad (3)$$

Equation 3 states that it should always be the case that each agent never review more than five papers. The $\mathcal{L}$: indicates that this is a *law policy*. The property *.reviews* can be considered as part of the system's state information. This is domain-specific and allows a more compact representation of the property. This policy may be easily represented by a finite automaton as shown in Figure 6.

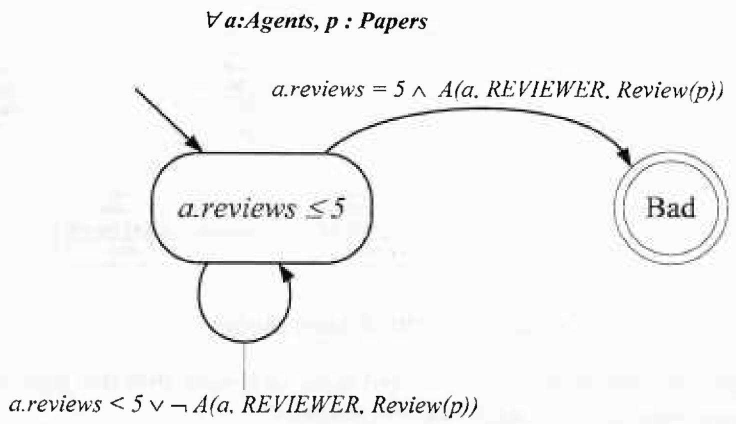

∀ a:Agents, p : Papers
a.reviews = 5 ∧ A(a. REVIEWER. Review(p))
a.reviews ≤ 5
Bad
a.reviews < 5 ∨ ¬ A(a. REVIEWER. Review(p))

Figure 6. No agent may review more than five papers

The use of the *A*() predicate in Figure 7 indicates an assignment of the *Reviewer* role to achieve the *Review paper* goal, which is parameterized on the paper *p*. This automaton depicts the policy in Equation 3, but in a manner for a model checker or some other policy enforcement mechanism to detect when violation occurs. The accepting state indicates that a violation has occurred. Normally, this automaton would be run alongside the system, either at design time with a model checker (Clarke Jr., Grumberg, & Peled, 1999), or at run-time with some policy enforcement mechanism (Ligatti, Bauer, & Walker, 2004).

We would like to emphasize here that we do not expect the designer to specify their policies by hand editing LTL. LTL is complex and designing policies in LTL would be very error prone and thus could potentially mislead the designer into a false sense of security or simply compose incorrect policies. There has been some work in facilitating the creation of properties in LTL (and other formalisms) for program analysis such as specification patterns (Dwyer, Avrunin, & Corbett, 1999). There has also been work done to help system property specification writers to graphically create properties (Smith, Avrunin, Clarke, & Osterweil, 2002) (backed by LTL) by manipulating automata and answering simple questions regarding elements of the property.

2.4.2 Law Policies Defined

The traditional notion of a policy is a rule that must always be followed. We refer to these policies as *law policies*. An example of a law policy with respect to our conference management example would be *no agent may review more than five papers*. This means that our system can never assign an agent to the *Reviewer* role more than five times. A law policy can be defined as:

[1]We only reason about bounded liveness properties because we only consider successful traces.

$$\mathcal{L} : Conditions \rightarrow Property \tag{4}$$

Conditions are predicates over state properties and events, which, when held true, imply that the *Property* holds true. The *Conditions* portion of the policy may be omitted if the *Property* portion should hold in all conditions, as in Equation 3.

Intuitively, for the example above, no trace in the system may contain a sub-trace in which an agent is assigned to the *Reviewer* role more than five times. This will limit the number of legal traces in the system. In general, *law policies reduce the number of legal traces for a multiagent system*. The policy to limit the number of reviews an agent can perform is helpful in that it will ensure that our system does not overburden any agent with too many papers to review. This policy as a pure law policy, however, could lead to trouble in that the system may no longer be able to achieve its goal. Imagine that more papers than expected are submitted. If there are not sufficient agents to spread the load, the system will fail since it is cannot assign more than five papers to any agent. This is a common problem with using only law policies. They limit the *flexibility* of the system, which we define as *how well the system can adapt to changes* (Robby et al., 2006).

2.4.3 Guidance Policies Defined

While the policy in Equation 3 is a seemingly useful policy, it reduces flexibility. To overcome this problem, we have defined another, weaker type of policy called *guidance policies*. Take for example the policy used above, but as a *guidance policy*:

$$\forall a_1 \text{: } Agents,\ \mathcal{G} : \Box(sizeOf(a_1.reviews) \leq 5) \tag{5}$$

This is the same as the policy as in Equation 3 except for the $\mathcal{G}$:, which indicates that it is a *guidance policy*. In essence, the formalization for guidance and law policies are the same, the difference is the intention of the system designer. *Law policies* should be used when the designer wants to make sure that some property is always true (e.g. for safety or security), while *guidance policies* should be used when the designer simply wants to guide the system. This guidance policy limits our agents to reviewing no more than five papers, *when possible*. Now, the system can still be successful when it gets more submissions than expected since it can assign more than five papers to an agent. When there are sufficient agents, the policy still limits each agent to five or fewer reviews.

Guidance policies more closely emulate how policies are implemented in human societies. They also provide a clearer and simpler construct for more easily and accurately describing the design of a multiagent organization. In contrast to policy resolution complexity of detecting and resolving policy contradictions in (Bradshaw et al., 2003), our methodology of using guidance policies present an incremental approach to policy resolution. That is, the system will still work under conflicting policies; its behaviors are amenable to analysis, thus allowing iterative policy refinement.

In the definition of *guidance policies*, we have not specified how the system should choose which guidance policy to violate in a given situation. We propose a partial ordering of guidance policies to allow the system designer to set precedence relationships between guidance policies. We arrange the guidance policies as a lattice, such that a policy that is a parent of another policy in the lattice, is *more-important-than* its children. By analyzing a system trace, one can determine a set of policies that were violated during that trace. This set of violations may be computed by examining the policies and checking for matches against the trace. When there are two traces that violate policies with a common ancestor, and one (and only one) of the traces violate the common ancestor policy, we mark the trace violating that common ancestor policy as illegal. Intuitively, this trace is illegal because the system could have violated a less important policy. Thus, if the highest policy node violated in each of the two traces is an ancestor of every node violated in both traces, and that node is not violated in both traces, then we know the trace violating that node is illegal and should not have happened.

Table 3 . Conference Management Policies

Node	Definition
P_1	No agent should review more than 5 papers.
P_2	PC Chair should not review papers.
P_3	Each paper should receive at least 3 reviews.
P_4	An agent should not review a paper from someone whom they wrote a paper with.

Take, for example, the four policies in the Table 3. Let these policies be arranged in the lattice shown in Figure 7. The lattice in Figure 7 means that policy P_1 is more important than P_2 and P_3, and P_2 is more important than P_4. Thus, if there is any trace that violates any guidance policies other than P_1 (and does not violate a law policy), it should be chosen over one that violates P_1.

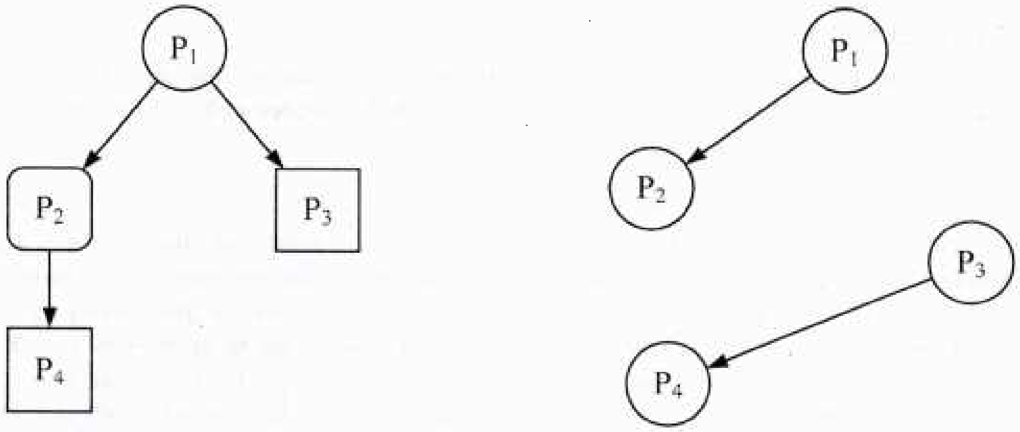


Figure 7. Partial orders of guidance policies (a) Possible Partial order of Guidance Policies (b) Another possible ordering

When a system cannot achieve its goals without violating policies, it may violate guidance policies. There may be traces that are still illegal, though, depending on the ordering between policies. *For every pair of traces, if the least upper bound of the policies violated in both traces, let us call this policy violation $\mathcal{P}$, is in one (and only one) of the traces, the trace with $\mathcal{P}$ is illegal.* For example, consider the ordering in Figure 7 let trace t_1 violate P_1 and P_2, while trace t_2 violates P_2 and P_3. Round nodes represent policies violated in t_1, box nodes represent policies violated in t_2, and boxes with rounded corners represent policies violated in both t_1 and t_2. Since P_1 is the least upper bound of P_1, P_2, and P_3 and since P_1 is not in t_2, t_1 is illegal.

As shown in Figure 7, the policies may be ordered in such a way that the policy violations of two traces do not have a least upper bound. If there is no least upper bound, $\mathcal{P}$, such that $\mathcal{P}$ is in one of the traces, the two traces cannot be compared and thus both traces are legal. The reason they cannot be compared is that we have no information about which policies are more important. Thus, either option is legal. It is important to see here that all the guidance policies do not need to be ordered into a single lattice. The system designer could create several unrelated lattices. These lattices then can be iteratively refined by observing the system behaviors or by looking at metrics generated for a certain policy set and ordering (e.g., (Robby et al., 2006)). This allows the system designer to influence the behavior of the system by making logical choices as to what paths are considered better. Using the lattice in Figure 7, we may even have the situation where P_1 is not violated by either trace. In this case, the violation sets cannot be compared, and thus, both traces are legal. In situations such as these, the system designer may want to impose more ordering on the policies.

Intuitively, guidance policies constrain the system such that at any given state, transitions that will not violate a guidance policy are always chosen over transitions that violate a guidance policy. If guidance policy violation cannot be avoided, a partial ordering of guidance policies is used to choose which policies to violate.

2.4.4 Effectiveness of Guidance Policies

2.4.4.1 CRFCC

Using our CRFCC example and a modified simulator from (Robby et al., 2006), we collected results running simulations with the guidance policy: *no agent should vacuum more than three rooms*. We contrast this with the law policy: *no agent may vacuum more than three rooms*. The guidance policy is presented formally in Equation 6.

$$\forall a_1: Agents,\ \mathcal{G}: \Box(vacummedRooms \leq 3) \tag{6}$$

For this experiment, we used five agents each having the following capabilities: a_1, org, search, and move; a_2, search, move, and vacuum; a_3, vacuum and sweep; a_4, sweep and mop; and a_5, org and mop. These capabilities restrict the roles our simulator can assign to particular agents. For example, the Organizer role may only be played by agent a_1 or agent a_5, since those are the only agents with the *org* capability. In the simulation we randomly choose capabilities to fail based on a probability given by the *capability failure rate*.

For each experiment, the result of 1000 runs at each capability failure rate was averaged. At each simulation step, a goal being played by an agent is randomly achieved. Using the capability failure rate, at each step, a random capability from a random agent may be selected to fail. Once a capability fails it cannot be repaired.

Figure 8 shows that while the system success rate decreases when we enforce the law policy, it does not, however, decrease when we enforce the guidance policy. Figure 9 shows the total number of times the system assigned vacuuming to an agent who already vacuumed at least 3 rooms for 1000 runs of the simulation at each failure rate. With no policy, it can be seen that the system will in fact assign an agent to vacuum more than 3 rooms quite often. With the guidance policy, however, the extra vacuum assignments (>3) stay minimal. The violations of the guidance policy increase as the system must adapt to an increasing failure of capabilities until it reaches a peak. At the peak, increased violations do not aid in goal achievement and eventually the system cannot succeed even without the policy. Thus, the system designer may now wish to purchase equipment with a lower rate of failure, or add more redundancy to the system to compensate. The system designer may also evaluate the graph and determine whether the cost of the maximum number of violations exceeds the maximum cost he is willing to incur, and if not, make appropriate adjustments.

2.4.4.2 Conference Management System

We also simulated the conference management system described in Section 2.3.1. We held the number of agents constant, while increasing the number of papers submitted to the conference. The system was constructed with a total of 13 agents, 1 *PC Member* agent, 1 *Database* agent, 1 *PC Chair* agent, and 10 *Reviewer* agents. The simulation randomly makes goals available to achieve, while still following the constraints imposed by GMoDS. Roles that achieve the goal are chosen at random as well as agents that can play the given role. The policies are given priority using the *more-important-than* relation as depicted in Figure 7.

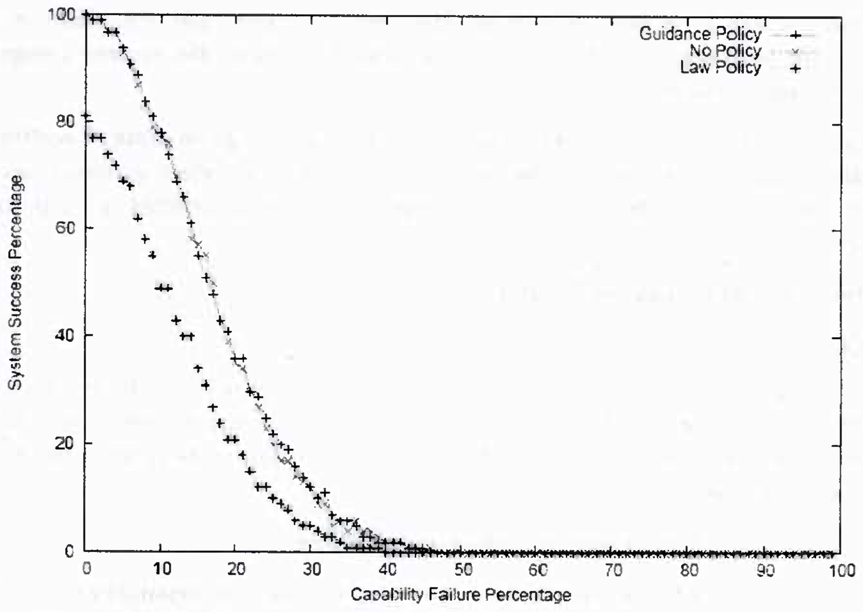


Figure 8 . The success rate of the system given capability failure

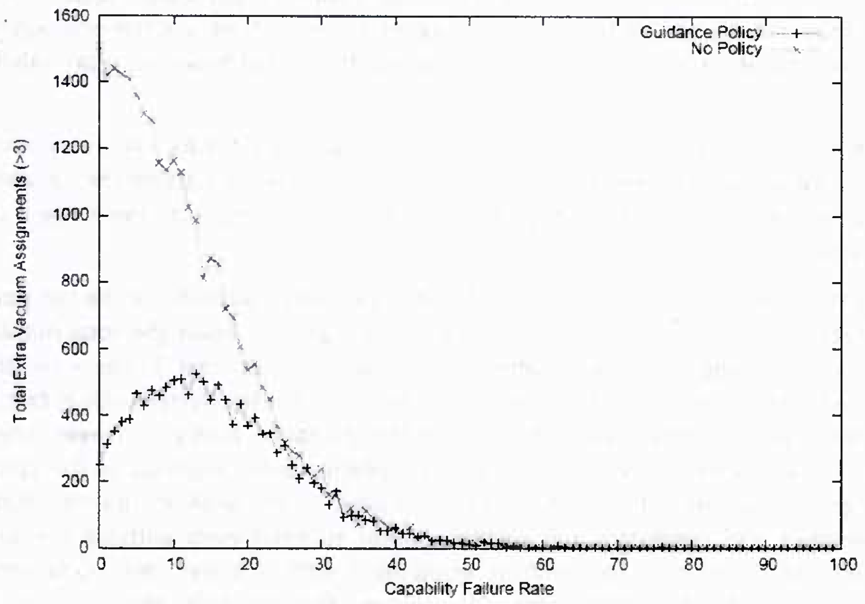


Figure 9 . The extra vacuum assignments given capability failure

Figure 10 shows a plot of how many times a guidance policy is violated versus the number of papers submitted for review. For each set of paper submissions (from 1 to 100) we ran the simulation 1000 times and then took the average of the 1000 runs to determine the average number of violations. In all the runs the system succeeded in achieving the top-level goal.

As seen by the graph in Figure 10, no policies are violated until around 17 papers (this number is explained below). The two least important policies (P_2 and P_3) are violated right away. The violation of P_2, however, levels off since it is interacting with P_1. The violations of P_3 is seen to grow at a much greater rate since it is the least important policy.

We then changed all the guidance policies to law policies and re-ran the simulation. For 17 or more submissions, the system always failed to achieve the top-level goal. This makes sense because we have only 10 Reviewer agents and we have the policies: the PC Chair should not review papers and no agent should review more than 5 papers. This means the system can only produce 5×10=50 reviews. But, since we have the policy that each paper should have at least 3 reviews, 17 submissions would need 17×3=51 reviews. For 16 or fewer papers submitted, the law policies perform identical to the guidance policies.

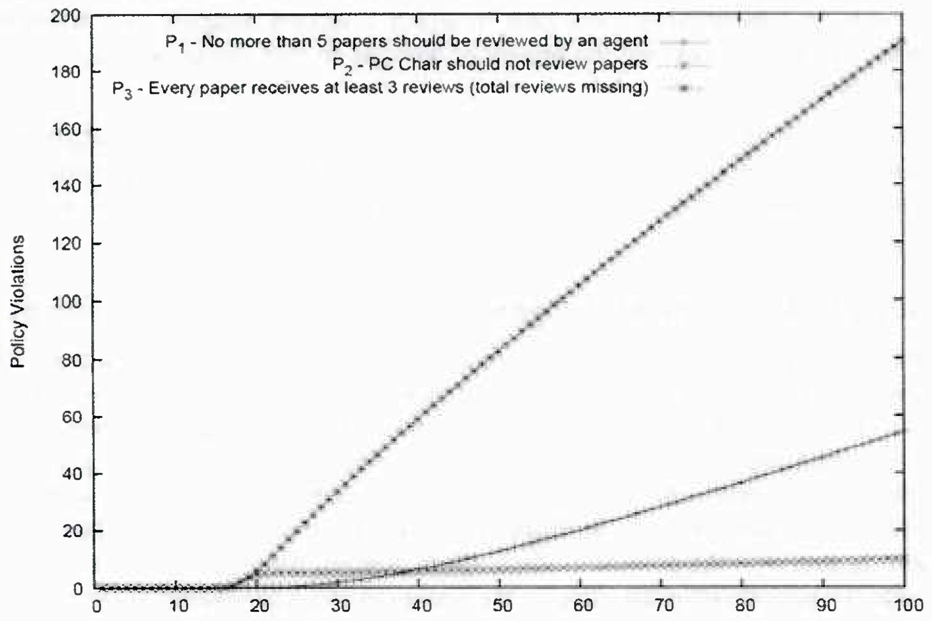


Figure 10. Violations of the guidance policies as the number of papers to review increases

2.4.4.3 Common Results

As the experimental results in Figure 8 show, guidance policies do not decrease the flexibility of a system to adapt to a changing environment, while law policies do decrease the flexibility of a system to adapt to a changing environment. Guidance policies, however, do help guide the system and improve performance as shown in Figure 9 and Figure 10. The partial ordering using the more-important-than relation helps a system designer put priorities on what policies they consider to be more important and helps the system decide which policies to violate in a manner consistent with the designer's intentions.

2.4.5 Conclusions

Policies have proven to be useful in the development of multiagent systems. However, if implemented inflexibly, situations such as described in (Peña, Hinchey, & Sterritt, 2006) will occur (a policy caused a spacecraft to crash into an asteroid). Guidance policies allow a system designer to guide the system while giving it a chance to adapt to new situations.

With the introduction of guidance policies, policies are an even better mechanism for describing desired properties and behaviors of a system. It is our belief that guidance policies more closely capture how policies work in human organizations. Guidance policies allow for more flexibility than law policies in that they may be violated under certain circumstances. In this paper, we demonstrated a technique to resolve conflicts when faced with the choice of which guidance policies to violate. Guidance policies, since they may be violated, can have a partial ordering. That is, one policy may be considered more important than another. In this manner, we allow the system to make better choices on which policies to violate. Traditional policies may be viewed as *law policies*, since they must never be violated. Law policies are still useful when the system designer never wants a policy to be violated–regardless of system success. Such policies might concern security or human safety.

Policies may be applied in an OMACS system by constraining assignments of agents to roles, the structure of the goal model for the organization, or how the agent may play a particular role. Through the use of OMACS, the metrics described in (Robby et al., 2006), and the policy formalisms presented here, we are able to provide an environment in which a system designer may formally evaluate a candidate design, as well as evaluate the impact of changes to that design without deploying or even completely developing the system.

Policies can dramatically improve run-time of reorganization algorithms in OMACS as shown in (Zhong & DeLoach, 2006). Guidance policies can be a way to achieve this run-time improvement without sacrificing system flexibility. The greater the flexibility, the better the chance that the system will be able to achieve its goals.

2.5 Learning to Self-Tune Multiagent Systems via Guidance Policies

Learning from mistakes is one of the most common learning methods in human society. Learning from mistakes allows one to improve performance over time, this is commonly referred to as *experience*. Experience can allow proper tuning and configuration of a complex multiagent system. In this paper, we implement this tuning and configuration through the mechanism of organizational guidance policies. Guidance policies are policies that have been defined to constrain the system, without limiting the system's flexibility (Harmon, DeLoach, & Robby, 2007). These policies may be suspended if the system cannot achieve its goal with the policies in place. These policies, however, still limit the system and can be used to *guide* the system while still allowing the system to adapt to changes in the environment.

Applying learning in multiagent systems is not new. Many authors have explored applying various learning techniques in a multiagent context (Jong & Stone, 2007, Nair, Tambe, Yokoo, Pynadath, & Marsella, 2003, Panait & Luke, 2005). Most of these learning applications, however, have been limited to improving an agent's performance at some task, but not the overall organization's performance. Some consider the overall team performance, but not in the structure of modern organization-based multiagent systems.

There is good reason past literature has not explored much polices over the entire organization. Reasoning over an entire multiagent system is a momentous task (Bernstein, Givan, Immerman, & Zilberstein, 2002). Care must be taken to ensure that all learned policies do not negatively impact the organization, for example, putting the organization in a situation where it is impossible for it to complete its goal. Interactions between policies can be subtle and many hidden interactions may exist. Our use of guidance policies (Harmon et al., 2007) helps mitigate these risks substantially, thus allowing the system to experiment with different policies without much risk to the viability of the system.

2.5.1 Self-Tuning Mechanism

An overview of how the learning takes place in our systems is given in Figure 11. The system first checks the goals that are available to be worked on, these goals come from GMoDS using the rules of precedence and trigger notations and support the overall goal of the system. Assignments of goals and the roles that can achieve them are made to the agents. The agents then indicate achievement failure or success. If an agent fails to achieve a goal, policies are learned over the current state knowledge. This cycles until either the system cannot achieve the main goal (system failure), or the system achieves the main goal (system success). Agents may fail to achieve a goal due to some aspect of the goal or due to some changes in the environment. The failure may be intermittent and random.

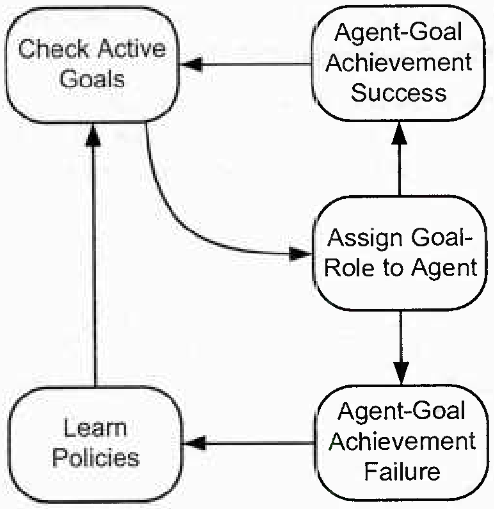

Check Active Goals
Agent-Goal Achievement Success
Assign Goal-Role to Agent
Learn Policies
Agent-Goal Achievement Failure

Figure 11. Learning Integration

The aim of our learning algorithm is to discover new guidance policies in order to avoid 'bad states'. Thus we will be generating only negative authorization policies. While the precise definition of a *bad state* is domain specific, we assume that the set of bad states is a possibly empty subset of all the states in the system. The remaining states are *good states*. Thus we have $S = S_B \cup S_G$ and $S_B \cap S_G = \emptyset$ where S is the set of all states in the system, S_G is the set of all good states in the system, and S_B is the set of all bad states in the system. Generally, bad states are states of the system that should be avoided. We use a scoring mechanism to determine which states are considered bad. The score of a state is domain specific, however, for our experiments we used

$$score(S_i) = \frac{1}{1 + F_i}$$

where F_i is the number of agent goal achievement failures thus far in state S_i.

To actually generalize and not simply memorize, our learning algorithm should derive policies that apply to more than one state through generalization. Thus, the learner attempts to discover the actual cause for the bad state so that it may avoid the cause and not simply avoid a single bad state.
Our learning algorithm takes a Q-Learning (Watkins, 1989) approach. The learner keeps track of both *bad* and *good* actions (transitions) given a state.

Bad-actions. *Bad actions* of a state are defined as actions that result in a new state that has a score lower than the state in which the action took place.

Good-actions. *Good actions* of a state are defined as actions that result in a new state that has a score equal to or greater than the score of the state in which the action took place.

The learner can then generate a set of policies that will avoid the bad states (by avoiding the bad action leading to this state). In the context of the experiments presented in this paper, the actions considered are *agent goal assignments* as defined in Section 1.3.3.1.

Figure 12 shows an example state transition. In this example, state S_i is transitioning to state S_{i+1} via assignment *T*. In state S_{i+1} a failure occurs, thus *F'*=*F*+1. The state is then given a score.

Generalization of the policies is done over the state. For each action leading to a bad state, the algorithm first checks to see if we already have a policy covering this action and the pre-state (state leading to the bad state), if so, nothing needs to be done for this bad action, otherwise we generate a policy for it. The algorithm generalizes the state by considering *matchings*.

Matching. A *matching* is a binary relation between a sub-state and a state. $B_k \prec S_i$, means that B_k matches S_i. Intuitively, a matching occurs between a state and sub-state when the sub-state represents some part of the state. Figure 13 depicts the sub-state matching relationship. Each letter represents a *state quantum*.

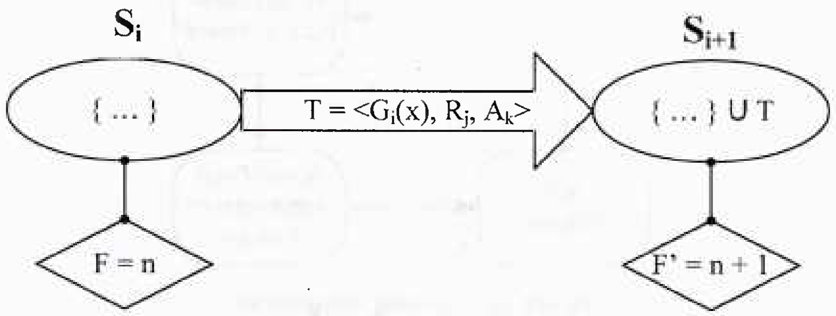


Figure 12. State and Action Transition

State Quantum. A *state quantum* is the smallest divisible (in terms of the algorithm) unit within a state.

In the unordered state, a matching sub-state may consist of any subset of state quanta. In the ordered state, the order of the quanta must be preserved in the sub-state. The empty sub-state is said to match *all* states. If a state is a set of unordered quanta, $S = \{s_1, s_2, \ldots\}$, then a sub-state, B_k, is said to be a matching for S_i iff $B_k \subseteq S_i$. In practice, since the number of ordered states for a given system is far greater than the number of unordered states for the same system, we tend to prefer using unordered states over ordered states. Using ordered states would give the learner more information, but in the experiments we conducted, that extra information was not worth the added state space. State space explosion with the ordered states was not offset by performance gains over the unordered version. In our experiments, we used agent role-goal assignment and achievement history as our states. The quanta are the actual agent assignment tuples. A sub-state is said to match a state when the agent assignment tuples in the achievement and assignment sets of the sub-state are subsets of the corresponding sets of the state. Intuitively, sub-states may be seen as generalizations of states.

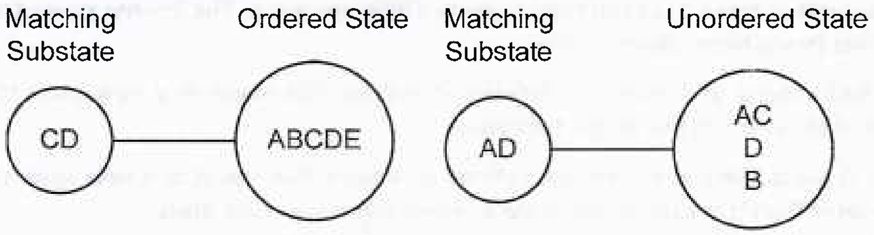


Figure 13. Ordered and Unordered states and their matching sub-states

The operation of the algorithm is independent of the state score calculation and the sub-state generation. For every bad action, *T*, given a state, the algorithm starting with the empty sub-state, which matches all states, computes a score using Equation 7. If this score is lower than a threshold, the algorithm asks the state for the set of smallest sub-states containing a specific sub-state (initially the empty sub-state). Each one of these sub-state-action pairs, (B_k, T), are given a score, $score(B_k, T) =$

$$1 - (size(match_G) / (size(states_G) + size(states_B) + size(match_B))) \qquad (7)$$

The variables in Equation 7 are as follows: $states_G$ is the entire set of good states given the transition T (gotten to by taking transition T); $states_B$ is the entire set of bad states given the transition T (gotten to by taking transition T); $match_G$ is, given the transition, the set of good states that the sub-state matches (a subset of $states_G$); and $match_B$ is, given the transition T, the set of bad states that the sub-state matches (a subset of $states_B$). It follows that the score is bounded as follows:

$$0 \leq score(B_k, T) \leq 1 \qquad (8)$$

It can easily be seen that when the sub-state matches all good states, and we have not encountered any bad states $size(states_G)=size(match_G)$ and $size(statesB)=size(matchB)=0$, thus $score(Bk,T)=0$. Conversely, if the sub-state does not match any good states, $size(matchG)=0$, thus score$(Bk,T)=1$. Each sub-state is scored with Equation 7. The sub-state with the highest score is chosen. If this score is less than a threshold constant, the process is repeated but the sub-state that is given to the state to generate new sub-states is this maximum score sub-state. Thus, we now build upon this sub-state and make it more specific.

Intuitively we start with the most general policy and then make it as specific as necessary (taking a greedy approach) so that we exclude 'enough' good states. The closer the threshold constant is to 1, the lower the possibility of the learned policies matching good states. The closer the threshold constant is to 0, the higher the possibility that the learned policies will match good states. For our experiments, we used a threshold constant of 0.6. This proved to perform sufficiently well for us since we dealt with intermittent failures. Pseudo-code for the policy generation is given Figure 14.

```
for all <action, preStateSet> in badTGivenS do
   for all preState in preStateSet}
      if !isAlreadyMatched(action, preState) then
         maxSubstate = emptySubstate
         maxScore = score(maxSubstate, action)
         done = false
         while maxscore < THRHLD ∧ !done do
            substateSet = getSubstates(preState, maxSubstate)
            if substateSet = Ø then
               done = true
            end if
            maxScore = -1
            for all substate in substateSet do
               score = score(substate,action)
               if score > maxScore then
                  maxScore = score
                  maxSubstate = substate
               end if
            end for
         end while
         matches = matches U <action, maxSubstate>
         policies = policies U generatePolicy(action, maxSubstate)
      end if
   end for
end for
```

Figure 14. Pseudo-code for generating the policies from the state, action pair sets

Another approach that we tested was to avoid the bad states, that is, ignoring transitions, simply construct policies that avoid generalizations (sub-states) of the bad states themselves. This alone can lead to problems. In our experiments, we found cases in which the system would live-lock. Since the learner never kept track of the event that lead to the bad state, it was possible that the learner might discover policies that forbid every agent but one from trying to achieve some goal. However, if this one agent failed with 100% probability, the system might simply reinforce that the new bad state was caused by the older decisions (sub-state) and would keep trying to assign the goal to the failing agent. The action-sub-state learner gets around this because the policies are developed for the action, if an action keeps leading to a bad state, there will be a policy discovered that forbids it. In the case where there are policies forbidding every agent from trying to achieve a goal, since we are using guidance policies, the policies will be suspended and an agent will be chosen. Eventually an agent who can achieve the goal will be chosen, and the learner will learn that that agent was not the cause of the previous failures. Avoiding live-lock is also the reason we regenerate policies after every failure.

2.5.2 Effectiveness of Self Tuning with Guidance Policies

To test our algorithm, we simulated three different systems: (1) a conference management system, (2) an information system, and (3) an improvised explosive device (IED) detection cooperative robotic system. These systems are a sampling across the usual multiagent system deployments: the conference management system, a human agent system; the information system, a software agent system; and the IED detection system, a robotic agent system. Each of these systems exhibits special tuning requirements given their different agent characteristics.

We simulated these systems by randomly choosing an active goal to achieve and then randomly choosing an agent capable of playing a role that can achieve the goal while following all current policies in the system. Active goals are simply goals in our goal model that GMoDS deems can be worked on. A goal may become active when triggered, or when precedence is resolved. In the case of learning, the learning code receives all events of the system. After every agent goal achievement failure, the system regenerates its learned guidance policies using all of the state, transition, and score information it has accumulated up to that point.

2.5.2.1 Conference Management System

In the Conference Management System example, we modified the agents such that some of them fail to achieve their goal under certain conditions. In this system we have three types of Reviewer agents: one that never fails when given an assignment, another that fails 70% of the time after completing two reviews (*Limit Reviewer*), and the last type, that fails 70% of the time when trying to review certain types of papers (*Specialized Reviewer*). We focused on failures of agents to achieve the *Review Paper* goal. We created the Reviewer types described above and observed the system performance. In this simulation, we varied the number of papers submitted for review to trigger the failure states. For our experiment, states contain the history of *assignments* and number of *failures* thus far. Actions are the *assignment* and *failure* events. Only leaf goals are used in assignments. Non-leaf goals are decomposed into the leaf goals. In this experiment, we are concerned with the *Reviewer* role. Several different agent types are capable of playing this role, although they clearly behave differently. Each role may achieve specific goals as shown in the Figure 4. We ran the system with no learning and recorded the number of failures. We then ran the system with the learning engaged and recorded the number of failures. Finally, we ran the system with hand-coded policies that we thought should be optimal and recorded the number of failures.

The policies generated by the learner consist of forbidding an action given a state matching. For example, the learner discovered the policy: given the empty sub-state (meaning in any state), forbid the agent assignment ⟨*Specialized Reviewer, PC Reviewer, Review papers (Theory)*⟩. Thus this policy applies to all states of the system and tries to avoid assigning theory papers to the *SpecializedReviewer*. Another policy discovered by the learner is given the sub-state *Assigned:*⟨*Limit Reviewer, PC Reviewer, Review* *papers (Theory)*⟩, forbid the action ⟨*Limit Reviewer, PC Reviewer, Review papers (Theory)*⟩. The learner must learn all permutations on the abstracted goal parameter. It is interesting here to note that the learner tries to forbid the *LimitReviewer* after just one successful assignment to it. This has the added benefit that the assignment fails as late as possible, thus the system or environment could have improved before a failure occurs. The number of policies generated is relatively small, staying less than 10 in all runs.

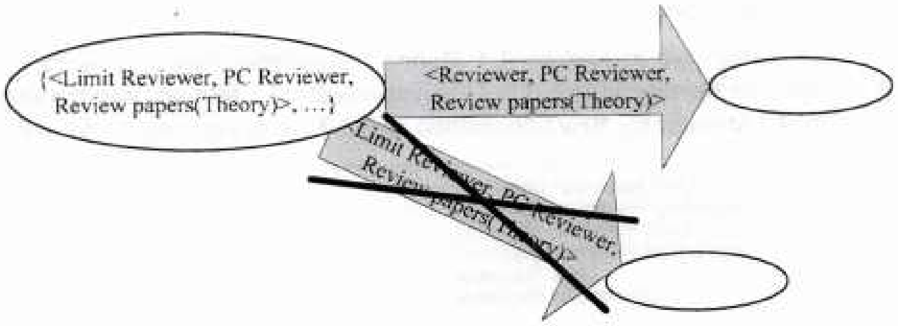


Figure 15 . Limit Reviewer policy preventing theory papers from being assigned

Figure 15 gives a graphical depiction of how a learned policy relates and is applied to the system states. From a state containing the sub-state, *Assigned:*⟨*Limit Reviewer, PC Reviewer, Review papers (Theory)*⟩, the assignment action ⟨*Limit Reviewer, PC Reviewer, Review papers (Theory)*⟩ is forbidden.

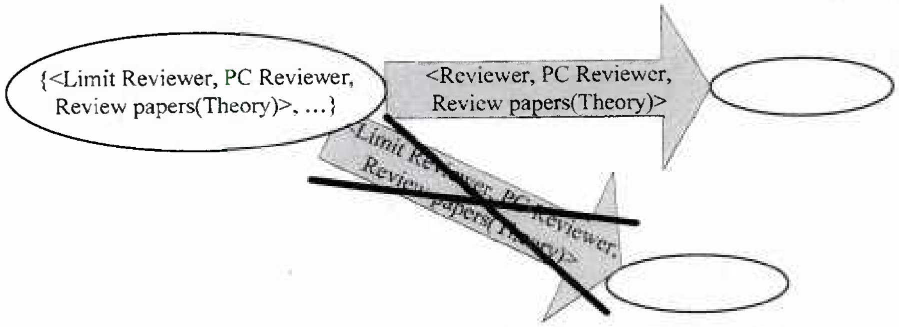


Figure 15 . Limit Reviewer policy preventing theory papers from being assigned

The agent type, role type, goal type, and a parameter abstraction is given. The parameter abstraction is currently domain specific, although a separate learner may categorize the parameters thus creating the abstraction function without the need for a domain expert.

Figure 16 compares the self-tuning, learning, system to one where this learning does not occur. We also compared the learning to a system, for which, a policy expert with knowledge of the agent goal achievement failure, hand coded policies he thought would tune the system. As can be seen, our learning algorithm does no worse than the hand-coded policies, and does vastly better than a non-tuned system. The learning tuning adapts the system quickly to its deployed environment—without requiring a policy expert to analyze the specific deployment environment and handcraft policies to tune for said environment.

2.5.2.2 Information System

Another multiagent system we tested was a simple Information System. Peer-to-peer information gathering and retrieval systems have been constructed using multiagent systems, e.g. Remote Assistant for Information Sharing (RAIS) (Mari, Poggi, Tomaiuolo, & Turci, 2006). In our information system we had four types of information retrieval agents: *CacheyAgent*, this agent fails with a 30% probability the first time it is asked for a certain piece of information, subsequent requests for info it has retrieved previously always succeeds; *LocalAgent*, this agent fails 100% of the time when asked for remote information, otherwise it succeeds with a 100% probability; *PickyLocalAgent*, this agent fails 100% of the time on any request except for on particular piece of local data; and *RemoteAgent*, this agent fails 100% of the time on all data except for remote data.

The leaf-goals of our system are as follows: 1.1 Monitor Information Search Requests, 1.2.1 Perform Local Search, 1.2.2 Perform Remote Search, 1.2.3 Present Results, 2.1 Monitor Information Requests, 2.2 Retrieve Information, 3.1 Monitor for New Information, and 3.2 Incorporate new Information into Index.

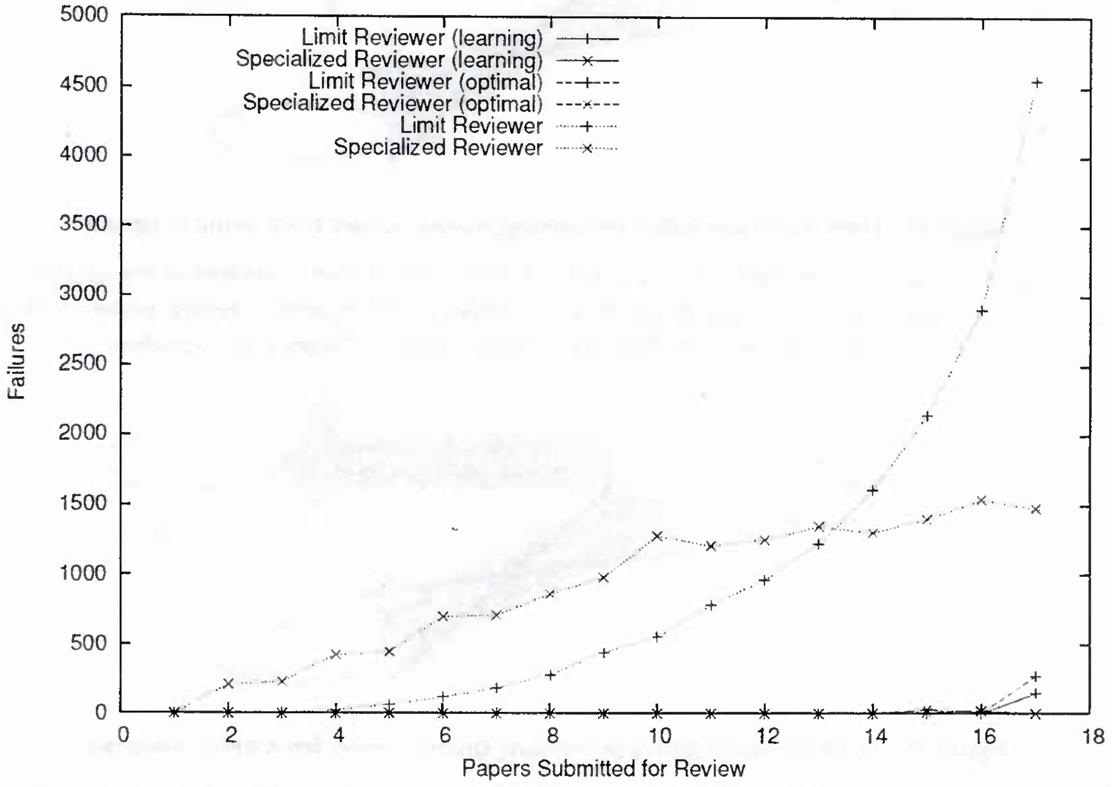


Figure 16. Agent goal achievement failures with self-tuning (learning), hand-tuning (optimal), and no tuning

The role-goal relation is shown in Table 4. *CacheyAgent* is capable of playing roles *Local Information Retriever* and *Remote Information Retriever*. *LocalAgent* and *PickyLocalAgent* are capable of playing *Local Searcher* and *Local Information Retriever* roles. *RemoteAgent* is capable of playing the *Remote Searcher* and the *Remote Information Retriever* roles.

The system quickly learned policies restricting the *PickyLocalAgent* from various information retrieval assignments. These policies were learned that they apply in all states. The *LocalAgent* also became restricted from being assigned any goal with any of the various remote information regardless of the system state. The policies concerning assignments to the *RemoteAgent* were similar to the *LocalAgent*. The *Cachey Agent*, however, was restricted by various policies, usually of the form ⟨*CacheyAgent, Remote Information Retriever, Retrieve Information (remote info 2)*⟩ agent goal assignment is forbidden given the state assignments contain ⟨*CacheyAgent, Local Information Retriever, Retrieve Information (*

local info 1)⟩. The learner did not have a mechanism for generating a negation policy. For example, do not allow the assignment if the state does *not* match the given sub-state. This could be a future enhancement to the policy generation of the learning. Although the learner did not have this capability, it still managed to keep the *Cachey Agent's* failures low, and the introduction of this type of agent did not confuse the learner with regards to the type of failure of the other agents.

Table 4 . IS Role Goal Relation

Role Name	Goals Achieved
GUI	1.1, 1.2.3, 2.1
Local Searcher	1.2.1
Remote Searcher	1.2.2
Information Monitor	3.1
Indexer	3.2
Local Information Retriever	2.2
Remote Information Retriever	2.2

Figure 17 shows the agent achievement failures for a non-tuned system. An expert may analyze these failures and craft policies to guide the system to avoid the failures, but this would be an error-prone and tedious task. In fact, by the time a solution is proposed, the problem may well have changed.

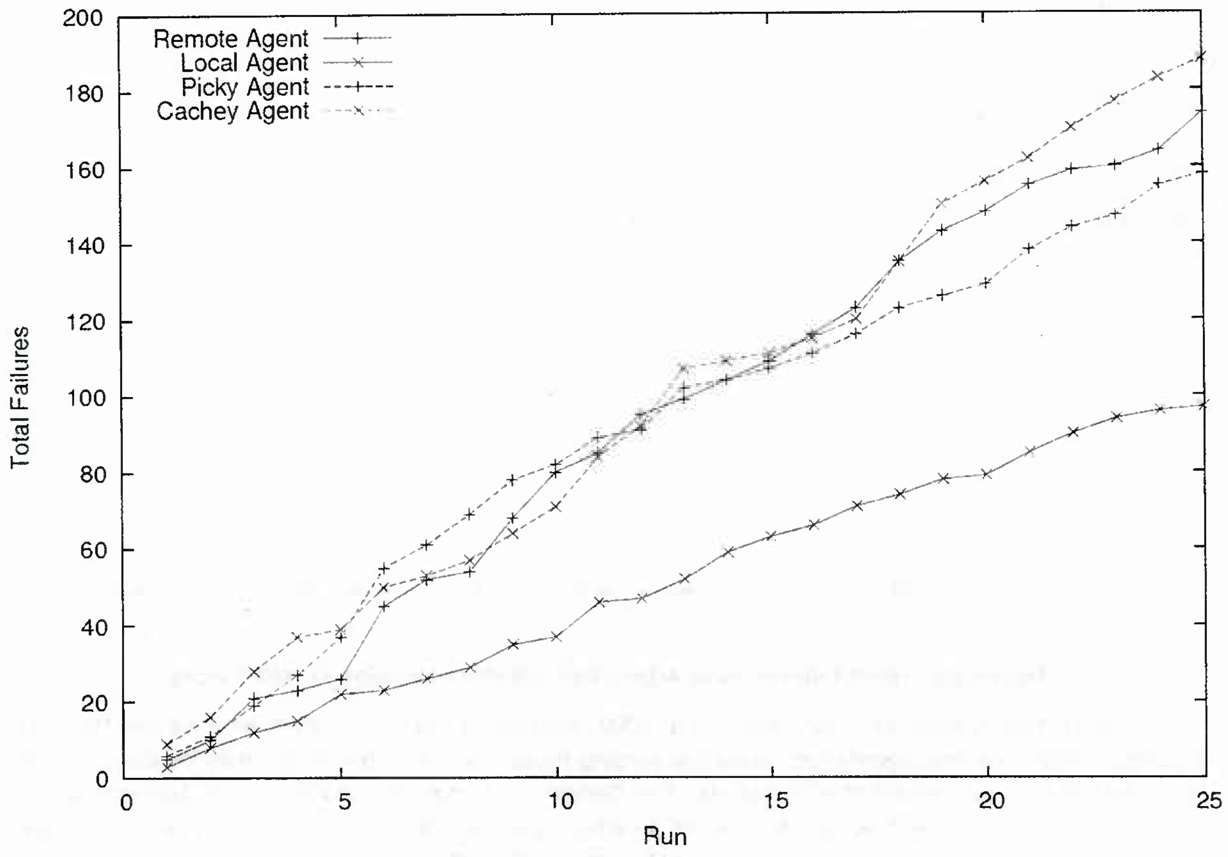


Figure 17 . Agent failures with no self-tuning

Our self-tuning learning achieved the results given in Figure 18. The system was able to self-tune and adapt itself to an environment that contained multiple unique causes of failure. The adaptation happened quickly enough to greatly benefit the system.

2.5.2.3 IED Detection System

The use of multiagent systems in robotic teams is a natural application of multiagent systems. Unfortunately, robots and their physical environment contain more variability than purely software-based multiagent systems. System designers may not be able to plan for all this potential variability when designing their system. Capabilities of robots may vary with their physical environment. For example, a robot needing line-of-sight for communication may go out of communication range if they move behind a physical structure in their environment. Other robots may find that they are not able to diffuse certain types of IEDs, perhaps because the IEDs are too big for the agent's grippers. Certainly, if the system designer could think of and design for all possible environmental variations, their system would be able to perform efficiently in all environments. In practice, however, this is not practical. Thus, the system should be able to automatically learn and adapt to environmental variations on its own.

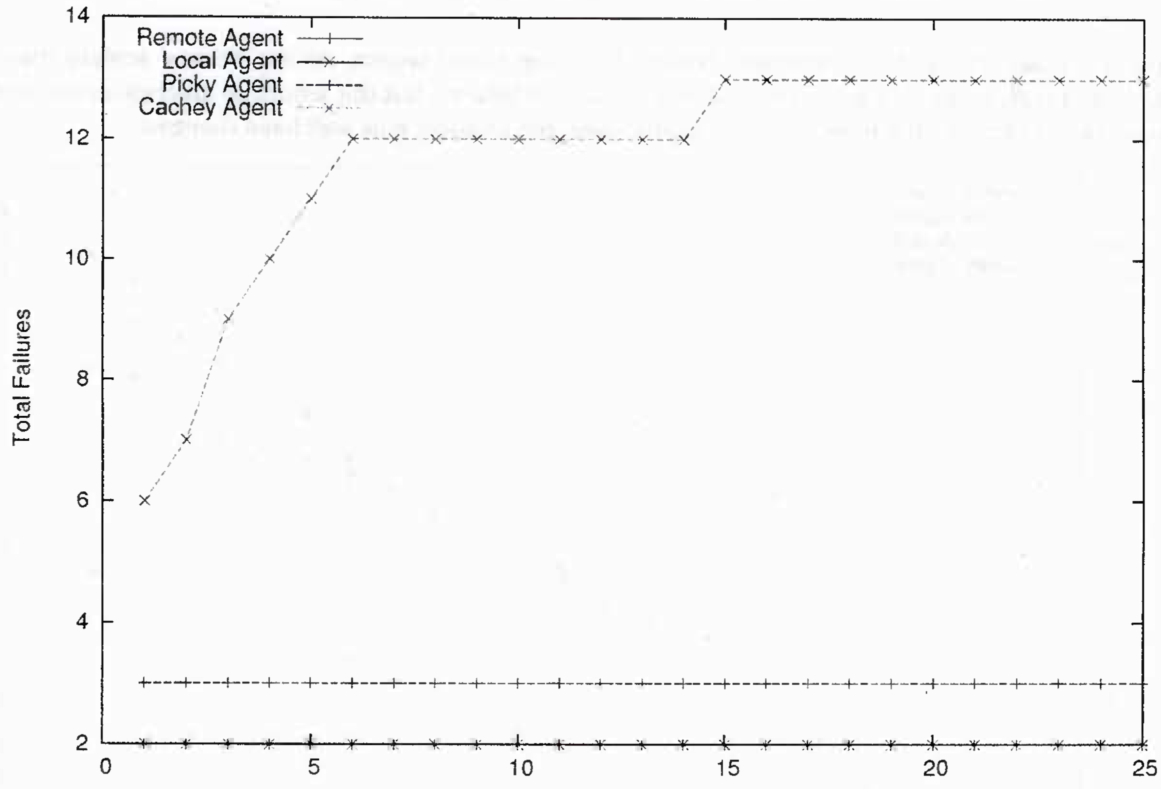


Figure 18 . Agent failures using action and sub-state learning for self-tuning

The IED detection system we used (MACR Lab, 2009) consists of agents to *Patrol* an area, *Identify* IEDs, and *Defuse* IEDs. Various agents are capable of playing these roles. The *Patrol* role may be played by our *LargePatroller* or our *SmallPatroller* agents. The *Defuser* role may be played by our *LargeGripper* or *SmallGripper* agents. IEDs may be found while an agent is playing the *Patrol* role, this event will trigger an Identify goal, which in turn could trigger a Defuse goal. The Defuse goal is parametrized on the type of IED identified (large or small). The IED patrol area is first broken into four parts as shown in Figure 19.

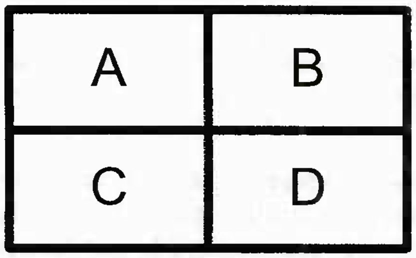


Figure 19. IED Search Area for various Patrollers

For the first experiment, we made the *ShortPatroller* fail on area *D* with a 40% probability for the first 10 assignments made to it. After the first 10 assignments to it, the *ShortPatroller* fails with a 40% probability on area *A* and no longer fails on area *D*. The *LargePatroller* agent always fails on area *B* with a 20% probability. The *SmallGripper* fails with a 100% probability on diffusing large IEDs.

Without learning, in the first 1000 runs, the *SmallGripper* failed a total of 14879 times, the *ShortPatroller* failed 409 times, and the *LargePatroller* failed 107 times. With learning, for the first 1000 runs, the *SmallGripper* failed 1 time, the *ShortPatroller* failed 4 times, and the *LargePatroller* failed 1 time. Subsequent runs using the accumulated knowledge displayed no failures by any agents.

In the second experiment, we left the agents the same except for the *LargePatroller*. The *LargePatroller* agent now fails on area *D* with a 20% probability. This overlaps with the failure area of the *SmallGripper* agent.

In this scenario, without learning, in the first 1000 runs the *SmallGripper* failed 15172 times, the *ShortPatroller* failed 434 times, and the *LargePatroller* failed 133 times. With learning, for the first 1000 runs, the *SmallGripper* failed 1 time, the *ShortPatroller* failed 3 times, and the *LargePatroller* failed 12 times. In a subsequent 1000 run, using the accumulated knowledge, the learning fell into a sub-optimum, the *SmallGripper* failed 0 times, but the *ShortPatroller* failed 2 times, and the *LargePatroller* failed 102 times. We hypothesize that this is due to the fact that we have the overlapping failing area as well as no partial ordering on learned guidance policies. Thus when the policies must be suspended due to conflicting, or because the system cannot progress with the policies, all the learned policies are suspended at once, creating the situation similar to having no learning.

2.5.2.4 Common Results

In all of our experiments, our self-tuning mechanism was able to quickly avoid multiple failure states that had multiple independent sources. The performance of the systems increased as the system tuned to its environment. The number of polices discovered was kept small which can be important when considering policies at run-time, since more policies can mean more processing time and effort.

In all of the scenarios, we had multiple independent failure vectors. The learning was able to overcome this. The IED simulation explored an evolving failure situation where the cause of the failure changed over time. The information system also had a unique type of failure with the *CacheyAgent*. This agent cached results of previous queries and thus would always succeed once it had successfully retrieved a particular piece of information; otherwise it had a certain probability of failure. This failure was handled by the algorithm, but due to the nature of the policies generated, it was not handled optimally and in a general sense.

2.5.3 Related Work

There has been much work in independent agent learning outside of the organizational framework. Much of this learning is on the individual agent level, with the hope that the over-all system performance will improve.

Bulka et al. (Bulka et al., 2007) devised a method allowing agents to learn team formation policies for individual agents using a Q-Learning and classifier approach. They showed notable improvement in the performance of their system. While their approach works well for open multiagent systems, it does not leverage the properties of an organization-based multiagent approach. Abdallah and Lesser (Abdallah & Lesser, 2007) developed a similar method to Bulka's except they were concerned with migrating the learned information to a changed network topology as well as using the learned information to optimize the network topology of the current agent system.

Kok and Vlassis (Kok & Vlassis, 2006) used coordination graphs along with a Q-Learning approach to learn to maximize overall agent coordination performance. Again, this work does not leverage the organizational approach to multiagent systems. Every agent is equal in society and only varies with respect to capabilities possessed. In human organizations, structures are built in which actors have roles that they can fill.

Other work has been done at the agent behavior level (e.g., Peshkin, Kim, Meuleau, & Kaelbling, 2001). As an agent tries to achieve a goal, it may affect the performance of its teammates. Policies are sometimes used to restrict the agent's behavior while working on a goal.

Chiang et al. (Chiang et al., 2007) have done some work on automatic learning of policies for mobile ad hoc networks. Their learning, however, was an offline approach using simulation to generate specific policies from general 'objectives' and possible configurations. Our research leverages the organizational framework to generate policies online that affect the system through the processes of the organization (i.e role-goal assignments).

2.5.4 Conclusions

Multiagent systems can become quite complex and may be deployed in environments not anticipated by the system designer. Furthermore, the system designer may not have the resources to spend on hand-tuning the system for a particular deployment. With these issues in mind, we have developed a method to create self-tuning multiagent system using guidance policies.

Using a variation of Q-Learning, we have developed an algorithm that allows the system to discover policies that will help maximize its performance over time and in varying environments. The use of guidance policies helps remove some of the traditional problems with using machine learning at the organization level in multiagent systems. If the learner creates bad policies, they should not prevent the system from achieving its goal (although they may degrade the quality or timeliness of the achievement). In this way, our approach is 'safer' and thus we can use a simpler learner.

In the experiments we conducted, the system was able to adapt to multiple agent goal achievement failures. It was able to cope with randomized and history-sensitive failures. The learner was able to discover guidance policies that, in every case, caused our systems to perform better on the order of a magnitude when faced with these failures.

Since we are taking the approach of always trying to avoid bad states, there is the question of whether our approach will possibly drive the system away from the optimal state in certain diabolical cases. The argument is that in order to get to the best state, we must pass through some bad states. To address this, we need to look at what is being considered as a bad state and if it is possible for there to be a high-scored state that can only be reached through a bad state. In the experiments we have performed, bad states corresponded to agent goal achievement failures. Using agent goal achievement failures as the scoring method of our states, it is possible that you must pass through a bad state in order to get to an end state with the highest score. But, since the score is inversely proportional to agent goal failures, we will have to go through a bad state in any case. We argue that since our score is monotonic, our algorithm should be able to drive toward the best-scored state even in the diabolical case that you must go through a bad state. This would require that we order our learned guidance policies using the more-important-than relation by score.

The usage of guidance policies allow for retention of useful preferences and automatic reaction to changes in the environment. For example, there could be an agent that is very unreliable, thus the system may learn a policy to not make assignments to that agent, however, the system may have no alternatives and thus must use this agent. "We don't like it, but we deal with the reality."–that is until another agent that can do the job joins the system.

2.6 Using Guidance Policies to Achieve Abstract Qualities

"Good, fast, or cheap, pick two." What drives designers to make decisions on how to architect a system? The stakeholder has certain abstract qualities in mind: efficiency, quality, reliability, and so forth. These qualities are desired by nearly every stakeholder, but how do we make sure our system is guided by these qualities? What happens when the system cannot provide all the qualities all the time? There will be a trade-off; some speed will be sacrificed for quality, or some quality for speed. This research describes how to analyze a design, automatically generate formal specifications in the form of policies, and put trade-offs into the open, allowing the decisions about what is more important to be made at design-time and not during implementation, which is now often the case.

2.6.1 Introduction

Organization-based multiagent systems engineering has been proposed as a way to design complex adaptable systems (Bernon et al., 2005, DeLoach et al., 2007). Agents interact and can be given tasks depending on their individual capabilities. The system designer is faced with the task of designing a system to not only meet function requirements, but also non-functional requirements. The system designer needs tools to help them generate specification and evaluate design decisions early in the design process. Conflicts within the non-functional requirements should be uncovered and, if necessary, the stakeholders' may then be consulted to help resolve those conflicts. The approach we are taking is to first generate a set of system traces using the models generated at design time. Second, we analyze the system traces, using additional information provided by the system designer. Third, we generate a set of policies that will guide the system toward the abstract qualities. And, fourth, we analyze the generated policies for conflicts.

2.6.2 Quality Metrics

ISO 9126 (ISO, 1991) defines a set of qualities for the evaluation of software. This document breaks down software qualities into six general areas: functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics are broad and may apply in different ways to different types of systems. In our research, we identified a group of metrics that evaluate multiagent system traces in order to illustrate our concepts. Each metric is formally defined and may be measured precisely over the current system design.

2.6.2.1 Efficiency

We are using a trace-length based definition of efficiency. The shorter the trace the more efficient is the top-level goal achievement. Our strategy here is to make assignments such that we minimize the total expected trace length. This minimization will be dynamic in that it will take into consideration the current trace progress. Thus we consider goal achievements the system has made when determining shortest trace length to pursue. We then convert this logic statically into a set of policies that when enforced will exhibit the same behavior as a minimization algorithm, given current system goal achievements. Given the initial system state, we prefer to make assignments to achieve the overall shortest path. Thus, we will be generating directing policies proscribing assignments (they may still have multiple options).

Expected trace length is defined in Equation 9.

$$ExpLength(\xi) = \sum_{i=1}^{n} \frac{1}{1-pf_i} \tag{9}$$

ξ represents a single trace, while pf_i is the probability of failure for the assignment achievement *i* within that trace. An *assignment achievement* is the accomplishment of an assignment by an agent. Thus, an *assignment achievement failure* is a failure of the agent to complete its assignment. The assignment achievement failure probability is defined in terms of capability failure given the assignment. The probability of failure for the agent goal assignment achievement (pf_i) is the maximum of the probability of failure for all the capabilities required for the role in the assignment:

$$pf_i = max_{c_x \in capreq(As_i)} PF(c_x, As_i) \tag{10}$$

It is evident here that some traces will have an infinite expected length (in the case of probability of failure is 100%).

We are concentrating on the assignment of goals (tasks) to agents in the system. A typical goal assignment choice is depicted in Figure 20. $G_1(x)$ represents the parametrized goal or task that needs to be achieved. R_1 and R_2 represent two different roles that are able to achieve the goal G_1. A_1 and A_2 represent two agents that posses the capabilities required to play each role. The failures are connected to each capability.

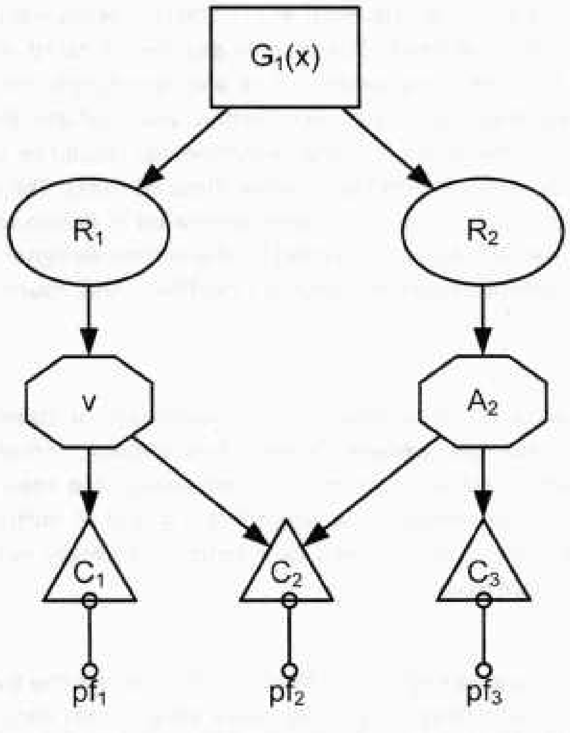


Figure 20. Goal Assignment Choice with Capability Failure Probabilities

The failure probabilities may be represented as a matrix as in Table 5. For a given capability, only certain portions of the assignment may be relevant to the probability of failure. In these cases, the matrix may be collapsed to a more compact representation.

Table 5 . Capability Failure Probabilities

Capability/Assignment	As_1	As_i	...
c_1	$pf(c_1,As_1)$	$pf(c_1,As_i)$	$pf(c_1,\ldots)$
c_2	$pf(c_2,As_1)$	$pf(c_2,As_i)$	$pf(c_2,\ldots)$
c_3	$pf(c_3,As_1)$	$pf(c_3,As_i)$	$pf(c_3,\ldots)$

2.6.2.2 Quality of Product

Quality of achieved goals falls under the ISO software quality of Functionality. Here we define quality as a measure of the quality of goal achievement. Quality of goal achievement may depend on the Role, Agent, and Goal Instance (including parameter). In our analysis, we limit ourselves to these influences although goal achievement quality may also depend on such things as environment, history of the system, and current assignments.

Certain roles may obtain a better result when achieving certain goals; likewise certain agents may play certain roles better than other agents. These properties can be known at design time. Usually, this intuition is known by the implementers, and they may manually design an agent goal assignment algorithm to favor these assignments.

For our analysis and experiments, we mapped assignments to scores. The higher the score, the higher the quality of product is. The scores can be specified over the entire assignment, or just portions. For example, role R_1 may achieve goal G_1 better than role R_2 when the parameter of G_1 is of class *x*. A designer could also specify agents who produce higher quality products and thus may be part of the score determination.

To analyze an entire trace, we score the trace by computing the average quality of product. We realize some products may be more important than others. Without loss of generality, however, we use a simple average as given in Equation 11 (the analysis could potentially include a more-important relationship between products). Let ξ_i be the *i*th agent goal assignment in trace ξ of length *n*.

$$Qual(\xi) = \sum_{i=1}^{n} \frac{score(\xi_i)}{n} \tag{11}$$

2.6.2.3 Reliability

Sometimes failure should be avoided at all costs, thus, even if there is a probabilistically shorter trace, it could be the case that we choose the longer trace because we want to minimize the chance of any failure. Formally, we want to minimize goal assignment failures. We do this by minimizing the probability of capability failure. Our strategy here is to pick the minimal failure trace given the current completed goals in the system.

We can use the capability failure matrix defined for Efficiency. The score we are trying to minimize is defined as:

$$Fail(\xi) = \sum_{i=1}^{n} pf_i \tag{12}$$

Where pf_i is defined as in the Efficiency metric (the probability of failure of assignment *i* within trace ξ).

It is important here to see the distinction between Reliability and Efficiency. Efficiency is concerned with minimizing the expected trace length, while Reliability is concerned with minimizing the total number of failures. Thus, if we are pursuing Efficiency, we may choose a path in which there may be some failures, even though there is a path with no failures, because the expected trace length is shorter in the path with failures. Reliability will always choose the path with less expected failures, even if the path is longer than another.

2.6.3 Policy Generation

To construct our policies, we first generate a set of traces using our OMACS models as input to our customized Bogor model checker. We then run a metric over each trace, giving it a score. The aim here is to create policies to guide the system to select the highest (or lowest) scoring trace, given any sub-trace prefix. Thus, we create a set of assignments that will guide the system toward the maximum scoring traces. There may be many traces with the same maximum score, in this case we have a set of options. This selection is illustrated in Figure 21. We generate policies that, when followed, guide the system to traces that look like the highest scoring traces. Thus, for the figure, we proscribe that from state S_1, you must make goal agent assignments that are in the highest scoring traces (S_2, S_2', etc). For every sub-trace, we generate a set of agent goal assignment options. This may lead to many policies. For this reason, after we generate the initial set of policies, we prune and generalize them.

Policies will be of the form:

$$[guard] \rightarrow \alpha_1 \vee \alpha_2 \vee \ldots \qquad (13)$$

where α_i is a generalized agent goal assignment and [*guard*] is a conditional on the state of the system. The guard is also specified in terms of the achieved agent goal assignments and the available goals. The guard notation above is simply syntactic sugar for the form $\alpha_1 \vee \alpha_2 \vee \ldots \vee \neg guardcondition$.

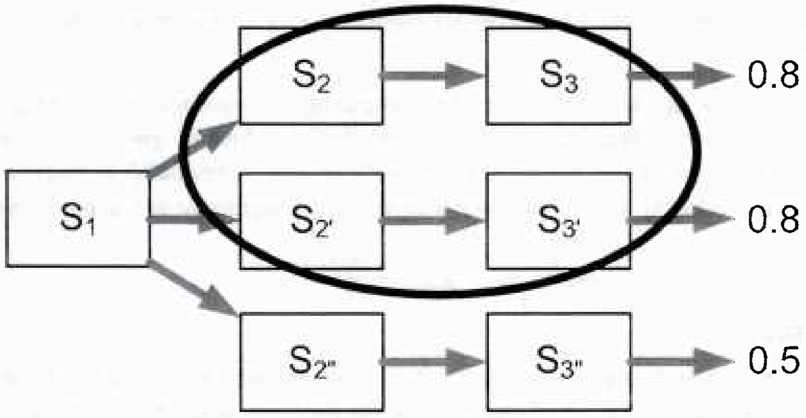


Figure 21. System Traces to Prefer

There are two different methods we use to reduce the size of the policy sets generated. One method is to prune useless policies. Since we produce a policy for every prefix trace, we may end up proscribing actions when the system did not initially even have a choice in the matter (the system was already forced to take that choice). We find these policies by checking the traces matched by the policies' left-hand side (the guard). If a policy only matches one trace, we can prune the policy as it had no effect on the system.

The second method combines multiple policies through generalization. If two or more policies offer the same choices for assignments (meaning their right hand sides are the same), the common pieces of the left hand side are computed. If the new left hand construct (the common pieces of the two policies) matches a non-empty intersection of traces with the current policies and the right-hand side of the new policy is not a subset of the right-hand side of each of the matching policies, the potential policy is discarded. Otherwise we remove the two policies that we have combined and add the new combined policy. This procedure is described more precisely in Figure 22. We repeat this procedure until we do not combine any more policies.

2.6.4 Conflict Discovery

Now that we are automatically generating policies for different abstract qualities, we may generate conflicting policies. These conflicts must be discovered. Once we discover these conflicts, we may use a partial ordering of policies to overcome these conflicts, or we may decide to even rework our initial system design.

For every quality, τ_n, we are generating a set of policies, P_n. The structure of a policy, ρ_i, in P_n is:

$$[guard] \rightarrow \alpha_1 \vee \alpha_2 \vee \ldots \qquad (14)$$

[*guard*] is the conditional which represents a generalization of the system state. Thus [*guard*] can be true over many different concrete states making the policy apply to many states of the system.

Since we have this policy structure and we are generating policies for different qualities, we may have different types of conflicts between policies from different *P*'s. These conflicts may be discovered by examining the guard of each policy and checking if there is an non-empty intersection of traces where both guards are active. If the intersection is non-empty and if the assignment choices are not equal, there is a possibility of conflict. Now, it may be the case that the right hand side of both AND'd together are satisfiable with the OMACS constraints, for instance, that only one agent may play a particular instance goal at once (Equation 15).

$$Sat((\alpha_x \vee \alpha_y \vee \alpha_z) \wedge (\alpha_a \vee \alpha_b \vee \alpha_c) \wedge \beta) \qquad (13)$$

Even if the equation is satisfiable, there may still be a need for partial ordering between the policies. Since there may be agent failure, we may want to know what policies to relax first.

We can partition the conflicts into two sets: *definitely conflicts* and *possibly conflicts*. The *definitely conflicts* elements will always conflict given the system design. The *possibly conflicts* will only conflict if the configuration of the system changes, i.e., in the case where there is capability or agent loss. If we have statistics on capability failure, we can even compute a probability of conflict. This probability could help the designer determine if the possibility of conflict is likely enough to spend more resources on overcoming it. Some conflicts may be inherent in the design, due to constraints on agents and capabilities or a sub-optimal configuration. Being able to see these conflicts as early as the design process (and especially before implementation) will greatly help, because it is much cheaper to change the design earlier rather than later.

If policies generated from different abstract qualities *definitely conflict*, then this is an indicator to the system designer that with the current constraints (agents, roles, and goals), it is not possible to satisfy all of the stakeholder's abstract requirements. The designer and/or stake holder must then decide to either modify the models by adding agents, changing goals, or by relaxing or redefining the abstract requirement.

We can, however, choose to resolve the conflicts by specifying which quality we prefer in each conflicting case. The designer may prefer efficiency over quality in certain cases and quality over efficiency in other cases. This choice will be a conscious decision by the designer (perhaps after

```
for all policy ∈ policies do
   combine = ∅
   for all policy2 ∈ policies do
      if policy ≠ policy2 ∧ policy.actions = policy2.actions then
         combine = combine ∪ policy2
      end if
   end for
   if combine ≠ ∅ then
      for all policy3 ∈ combine do
         commonGuard = policy.guard ∩ policy3.guard
         fail = false
         for all policyOld ∈ policies do
            if  policyOld.guard ⊆ commonGuard ∧ policyOld.actions
            ⊄ policy3.actions then
               fail = true
            end if
         end for
         if ¬ fail then
            policies = policies - {policy, policy3}
            policies = policies ∪ {<commonGuard, policy.actions>}
         end if
      end for
   end if
end for
```

Figure 22. Policy combining algorithm

consulting the stake-holders), and thus, a more engineered approach than the ad-hoc and unclear decisions that might be inadvertently made by implementers.

2.6.5 Effectiveness of Using Guidance Policies for Achieving Abstract Qualities

We took three different multiagent systems to test our policy generation. One example, the Cooperative Robotic Floor Cleaning Company (CRFCC) describes a scenario where a multiagent system is given the goal of cleaning the floor of a multifloor building. For the CRFCC system, we explored the efficiency quality. The second system we used is the Improvised Explosive Device (IED) Detection; in this example, we focus on the reliability of the system. The third example is an Information System. Quality of product was the concentration in that example.

2.6.5.1 CRFCC

We test our approach on the Cooperative Robotic Floor Cleaning Company (CRFCC) as described in Section 2.3.2. An example policy is given in Figure 23. The left side of the policy is the guard, in Figure 23, each of the agent goal assignments on the left hand side are AND'd together (they must all be true). The number after each agent goal assignment is the minimum number of agent goal assignment achievements that must have occurred thus far to make the assignment clause true. Each agent goal assignment on the right hand side is an option the system may take when choosing agent goal assignments in order to follow a minimum probabilistic trace.

<Agent2,Pickuper,Pickuparea()>:4		<Agent2,Pickuper,Pickuparea()>
<Agent1,Organizer,Dividearea()>:1	=>	<Agent5,Vacuummer,VacuumArea()
<Agent5,Vacuummer,VacuumArea()>:1		

Figure 23. CRFCC Generated Policy

We generated the policies and ran a simulation of the CRFCC system. The average of 1000 runs at each number of rooms was computed. We plotted the total number of assignments the system made to achieve its objective. This allows us to evaluate the impact of the policies.

Figure 24 gives the results for a system with the generated policies and one without the generated policies. The plot clearly shows that the system with the generated policies is guided toward efficiency, while the system without the policies has a much lower efficiency.

2.6.5.2 Information System

We also tested the Information System as described in Section 2.5.2.2. We generated a set of *quality* policies using the following properties: the *Remote Retrieval* role achieves a better quality of product than the *Local Retrieval* role for any goal parameter. However, the *PickyLocalAgent* performs the best when using the *Local Retrieval* role to achieve the *Retrieve Information* goal for any parameter. Using the automatic policy generation we generated 627 policies, which were than reduced to 69 using the techniques described in Section 2.6.3. We then ran simulations; the results showed that when the policies were enforced, we always achieved the highest quality results, without the sacrifice of any additional failures.

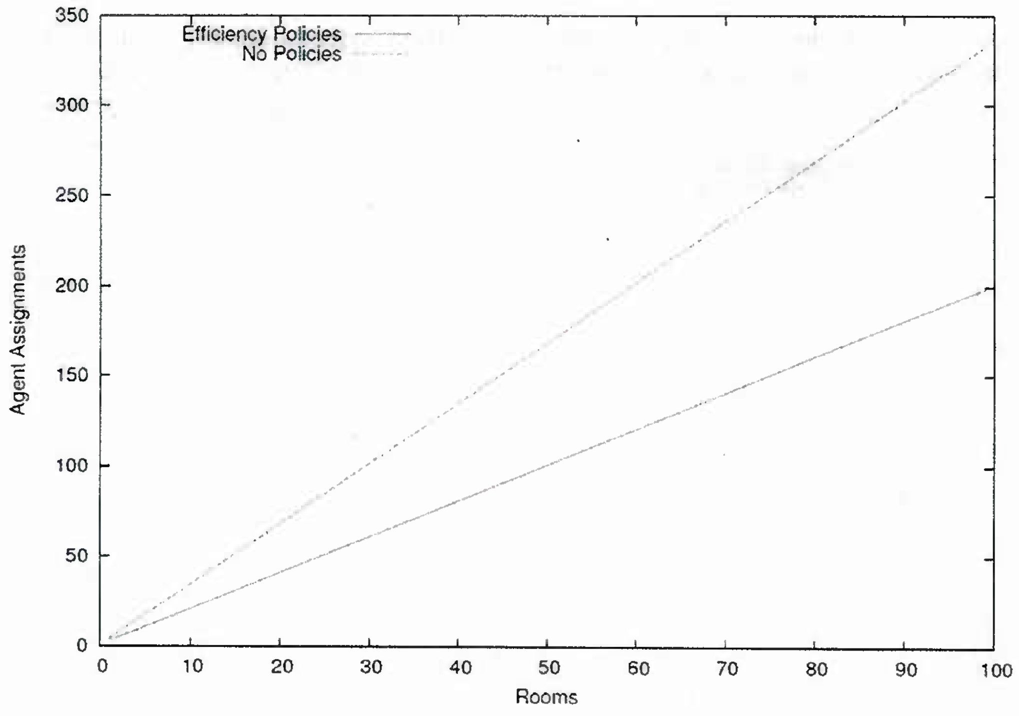


Figure 24. Efficiency Policy Generation Effects on Assignments

2.6.5.3 IED Detection System

We also tested our approach against the IED Detection System as described in Section 2.5.2.3. Various circumstances may cause an agent to fail or may allow an agent to succeed. For a system such as this, it is critical that it be as reliable as possible. We focus on this aspect in our policy generation. Table 6 gives the collapsed capability failure matrix for the IED detection system. Capabilities not listed have a 0 expected probability of failure. We generated policies by analyzing the traces generated over the models. The system initially generated 209 policies; after automatic pruning and generalization, we had 45 policies. We then ran the system using a simulator, breaking up the search area into pieces from 1 to

99. At each number of search areas, we ran the simulation 1000 times and took the average of the results.

Table 6. IED Goal Assignment Choice with Capability Failure Probabilities

Capability/Assignment	*
movement/SmallPatroller	0.5
dispose/SmallGripper	0.1

The policies forced the system to never have an agent failure. This is reflected in the graph in Figure 25. As can be seen, the number of agent assignments with the generated policies is less. This is because in the case of agent failures the goal (task) must be reassigned.

2.6.6 Related Work

There has been work in incorporating abstract qualities into multiagent systems. Tropos defines the concept of a soft-goal (Bresciani, Giorgini, Giunchiglia, Mylopoulos, & Perini, 2004) that describes abstract qualities for the system. These soft-goals, however, are mainly used to track decisions in the goal model design for human consideration.

Some work has been done on model checking multiagent systems (Robby et al., 2006, Viganò & Colombetti, 2008). While this work has helped the designer by providing some feedback on their design, it has not yet leveraged model checking to help automate the design process to the degree we present.

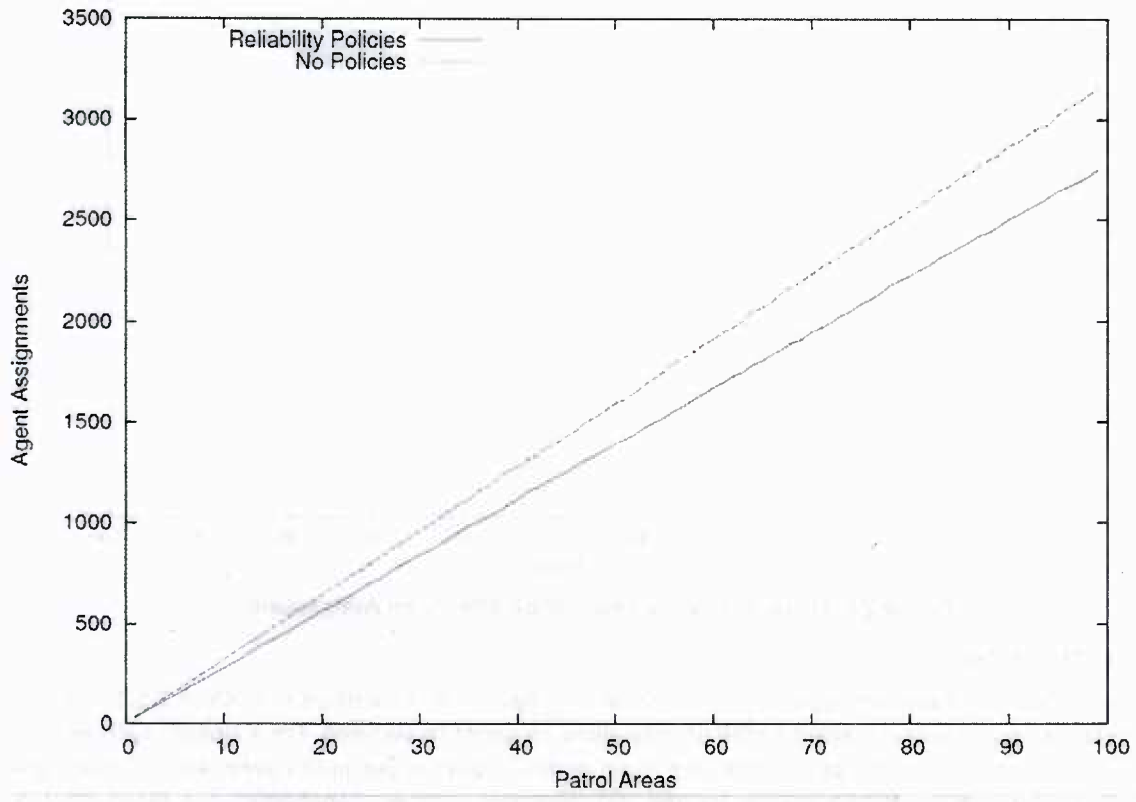


Figure 25. Reliability Policy Generation Effects on Assignments

2.6.7 Conclusions

We have provided a framework for stating, measuring, and evaluating abstract quality requirements against potential multiagent system designs. We do this by generating policies that can be used either as a formal specification, or dynamically within a given multiagent system to guide the system toward the maximization (or minimization) of the abstract quality constraints. These policies are generated offline, using model checking and automated trace analysis.

Our method allows the system designer to see potential conflicts between abstract qualities that can occur in their system. This allows them to resolve these conflicts early in the system design process. The conflicts can be resolved using an ordering of the qualities that can be parametrized on domain specific information in the goals. They could also cause a system designer to decide to change their design to better achieve the quality requirements. This is easier, since we are in the system design and not in the implementation.

These policies can be seen to guide the system toward the abstract qualities as defined in the metrics. Our experiments showed significant performance, quality, and reliability increases over a system using the same models, but without the automated policy generation.

2.7 Future Work

Guidance policies add an important tool to multiagent policy specification. However, with this tool comes complexity. Care must be taken to insure that the partial ordering given causes the system to exhibit the behavior intended. Tools that can visually depict the impact of orderings would be helpful to the engineer considering various orderings. We are currently working on inferring new policies from a given set of policies. For example, if a system designer wanted to get their system to a state for which they defined policy, we would automatically generate guidance policies. This could be useful when the policies are defined as finishing moves similar to finishing moves in chess. That is they proscribe optimal behavior, given a state. Thus, we would like to get to the state where we know that optimal behavior.

Guidance policies may be ordered using a *more-important-than* relation. In our policy learning work, we did not utilize that ordering. However, we hypothesize that if we ordered the learned policies by confidence, the system would be able to recover from learning errors more quickly. This is because the learning error policy would have a lower confidence than the other policies and thus would be suspended first in the case of a policy conflict or when the system cannot progress with the current policy set. Confidence may be computed using the score function described in Section 2.5.1.

Currently our learning algorithm assumes that only one action happens at a time. The actions of a multiagent system are not always easily serializable. In order to handle concurrent actions, we propose to consider the concurrent actions as a new compound action. In this way you may use this algorithm with minimum modification.

Another idea to improve the policy learning is by automatically abstracting the goal parameters and the state space using model checking to create a state space abstraction. We could then use this state space abstraction during learning instead of the exact state space. This would help with state space explosion and automate the goal parameter abstraction.

In all of our policy generation for abstract qualities experiments we abstracted away the goal parameter. Automated abstraction of the goal parameter using classification is an area that needs to be explored. If we had a more precise notion of the goal parameter, we would better be able to generate policies targeting attributes of the goal parameter. This would help in an automated policy conflict resolution suggestion.

The effect of applying multiple qualities within the same multiagent system should be evaluated. We expect that with conflict resolution, the designer will be able to prioritize the different qualities in different situations. This must be studied in current multiagent system designs to determine how much automation is required to feasibly resolve these conflicts.

Another area that can be explored is the splitting of policies to be able to target conflicts more precisely automatically. This of course needs to be balanced with generality (we would not want one policy for each possible scenario).

Currently our policy-combining algorithm does not ensure that we get the minimum number of policies possible given the previously generated policies. This is because we combine the policies two at a time randomly choosing with no backtracking. It could be the case that if we did not combine the first two policies, we may have been able to combine more later. In our experiments, this was not much of an issue since we were left with a small number of policies after combining (less than 100). This, however, could become an issue, as designs grow larger and should be considered.

More metrics over the multiagent system traces should be developed. For example, Security also falls under the ISO software quality of Functionality. Focusing on information flow, we could generate a set of policies to minimize information dissemination (maximizing privacy).

Policies show much promise as a specification tool for Multiagent systems. Systems must be adaptable, composable, and reliable. As technology grows more and more pervasive, so will the need to be able to verify that these systems will behave within certain bounds. This requires that we engineer our systems from the ground up with formalisms. These formalisms must describe the designer's requirements and intentions. We will also need a means of analysing the impacts of the designer's intentions. Using formal policies is one way of achieving this automated verification.

3 Organization-Based Multiagent Systems Engineering

Our main research objective was to develop a software engineering methodology for engineering high-assurance multiagent systems. Our goal was to provide a complete software engineering methodology for developing multiagent systems where assurance techniques are integrated in all levels of the development cycle starting from system design down to deployment.

This part of the report describes the Organization-based Multiagent System Engineering (O-MaSE) Process Framework, which helps process engineers define custom multiagent systems development processes. O-MaSE builds off the MaSE methodology and is adapted from the OPEN Process Framework (OPF). OPF implements a Method Engineering approach to process construction. The goal of O-MaSE is to allow designers to create customized agent-oriented software development processes. O-MaSE consists of three basic structures: (1) a metamodel, (2) a set of methods fragments, and (3) a set of guidelines. The O-MaSE metamodel defines the key concepts needed to design and implement multiagent systems. The method fragments are operations or tasks that are executed to produce a set of work products, which may include models, documents, or code. The guidelines define how the method fragments are related to one another. The paper also demonstrates two examples of creating custom O-MaSE processes.

3.1 Introduction

The software industry is facing new challenges. Businesses today are demanding applications that can operate autonomously, can adapt in response to dynamic environments, and can interact with other applications in order to provide comprehensive solutions. Multiagent system (MAS) technology is a promising approach to these new requirements (Luck, McBurny, Shohory, & Willmot, 2005). Its central notion – the intelligent agent – encapsulates all the characteristics (i.e., autonomy, proactive, reactivity, and interactivity) required to fulfill the requirements demanded by these new applications.

In order to develop these autonomous and adaptive systems, novel approaches are needed. In the last several years, many new processes for developing MAS have been proposed (Bergenti, Gleizes, & Zambonelli, 2004); unfortunately, none of these processes have gained widespread industrial acceptance. Reasons for this lack of acceptance include the variety of approaches upon which these processes are based (i.e., object-oriented, requirements engineering, and knowledge engineering) and the lack of Computer Aided Software Engineering (CASE) tools that support the process of software design. There have been some approaches suggested for increasing the change of industry acceptance. For instance, Odell et al. suggest presenting new techniques as an incremental extension of known and trusted methods (Odell, Parunak, & Bauer, 2001), while Bernon et al. suggest the integration of existing agent-oriented processes into one highly defined process (Bernon, Cossentino, Gleizes, Turci & Zambonelli, 2004). Although these suggestions may be helpful in gaining industrial acceptance of agent-oriented techniques, we believe that a more promising way is to provide more flexibility in the approaches offered. The main problem with these approaches is that they do not provide assistance to process engineers on how to extend or tailor these processes. In this vein, Henderson-Sellers suggests the use of method engineering using a well defined and accepted metamodel in order to allow users to construct and to customize their own processes that fit their particular approaches to systems development (Henderson-Sellers & Giorgini, 2005). Henderson-Sellers argues that by defining method fragments based on a common underlying metamodel, new custom processes can be created that support user defined goals and preferences.

This part of the report presents an overview of the Organization-based Multiagent System Engineering (O-MaSE) Process Framework. The goal of the O-MaSE Process Framework is to allow process engineers to construct custom agent-oriented processes using a set of method fragments, all of which are based on a common metamodel. To achieve this, we define O-MaSE in terms of a metamodel, a set of method fragments, and a set of guidelines. The O-MaSE *metamodel* defines a set of analysis, design, and implementation concepts and a set of constraints between them. The *method fragments* define how a set of analysis and design products may be created and used within O-MaSE. Finally, *guidelines* define how the method fragment may be combined to create valid O-MaSE processes, which we refer to as O-MaSE compliant processes.

3.2 Background

One of the major problems faced by agent-oriented software engineering is the failure to achieve a strong industry acceptance. One of the reasons hindering this acceptance is a lack of an accepted process-oriented methodology for developing agent-based systems. An interesting solution to this problem is the use of approaches that allow us to customize processes based on different types of applications and development environments. One technique that provides such a rational approach for the construction of tailored methods is Method Engineering (Brinkkemper, 1996).

Method Engineering is an approach by which process engineers construct processes (i.e., methodologies) from a set of method fragments instead of trying to modify a single monolithic, "one-size-fits-all" process. These fragments are generally identified by analyzing these "one-size-fits-all" processes and extracting useful tasks and techniques. The fragments are then redefined in terms of a common metamodel and are stored in a repository for later use. To create a new process, a process engineer selects appropriate method fragments from the repository and assembles them into a complete process based on project requirements (Brinkkemper, 1996).

However, the application of Method Engineering in the development of agent-oriented applications is non-trivial. Specifically, there is no consensus on the common elements of multiagent systems. Thus, it is has been suggested that prior to developing a set of method fragments, a well-defined metamodel of common agent-oriented that are typical of most varieties of MAS (e.g., adaptive, competitive, self-organizing, etc.) should be developed (Beydoun, Gonzalez-Perez, Henderson-Sellers, & Low, 2005).

Fortunately, we can leverage the OPEN Process Framework (OPF), which provides an industry-standard approach for applying Method Engineering to the production of custom processes (Firesmith & Henderson-Sellers, 2002). The OPF uses an integrated metamodel-based framework that allows designers to select method fragments from a repository and to construct a custom process using identified construction and tailoring guidelines. This metamodel-based framework is supported by a three-layer schema as shown in Figure 26. The M2 layer includes the OPF metamodel, which is a generic process metamodel defining the types of method fragments that can be used in M1. Thus a process (such as OPEN) can be created in M1 by instantiating method fragments from the M2 metamodel.

The OPF metamodel consists of Stages, Work Units (Activities, Tasks, and Techniques), Producers, Work Products, and Languages. A *Stage* is defined as a "formally identified and managed duration within the process or a point in time at which some achievement is recognized" (Firesmith & Henderson-Sellers, 2002, pp. 55). Stages are used to organize *Work Units*, which are defined as operations that are carried out by a *Producer*. There are three kinds of Work Units in OPF: Activities, Tasks, and Techniques. *Activities* are a collection of Tasks. *Tasks* are small jobs performed by one or more Producers. *Techniques* are detailed approaches to carrying out various Tasks. *Producers* use Techniques to create, evaluate, iterate, and maintain Work Products. *Work Products* are pieces of information or physical entities produced (i.e., application, document, model, diagram, or code) and serve as the inputs to and the outputs of Work Units. Work Products are documented in appropriate *Languages*.

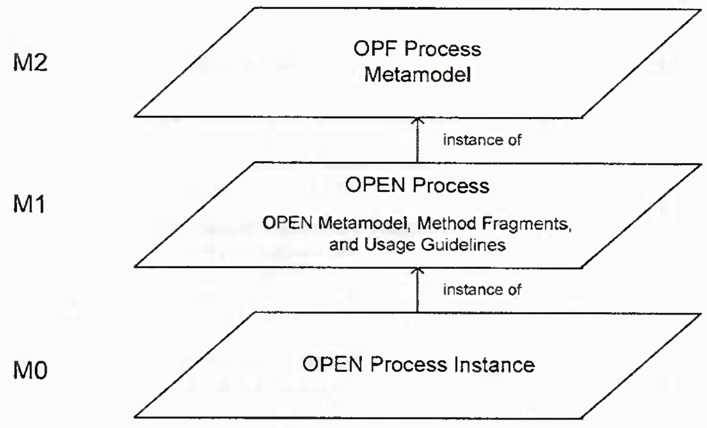


Figure 26. OPEN Process Framework

The M1 layer serves as a repository of method fragments instantiated from the M2 metamodel. A set of rules governing the relationship between these concepts (i.e., a process-specific metamodel and a set of reusable method fragments) is also defined in M1. The process engineer uses the guidelines to extend, to instantiate, and to tailor the predefined method fragments for creating a *custom process* in the M1 layer. These custom processes are then instantiated at the M0 level on specific projects; the actual custom process as enacted on a specific project is termed a *process instance*.

Alternatively, the FIPA (Foundation for Physical Agents) Technical Committee (TC) methodology group[2] is working on defining reusable method fragments in order to allow designers to specify custom agent-oriented processes (Seidita, Cossentino, & Gaglio, 2006). Although this approach is quite similar to OPF (they are both based on method engineering), its metamodel is derived from the Object Management Group (OMG) Software Process Engineering Metamodel[3] (SPEM). SPEM is based on three basic process elements that encapsulate the main features of any development process: Activities, Process Roles, and Work Products. Development processes are assembled from a set of SPEM Activities, which represent tasks that must be done. An Activity is essentially equivalent to an OPF Work Unit and is performed by one or more Process Roles (which corresponds to OPF Producers). Process Roles carry out the Activities in order to produce Work Products (the same term is used here by OPF). A detailed description of this metamodel and a comparison with other method fragment proposals can be found in (Cossentino, Gaglio, Henderson-Sellers & Seidita, 2006). The next section focuses on using Method Engineering and the OPF metamodel to specify O-MaSE.

3.3 O-MaSE Process Framework

In this section, we define the O-MaSE Process Framework as shown in Figure 27, which is analogous to the OPF from Figure 26. In fact, we use the OPF metamodel in level M2. Level M1 contains the definition of O-MaSE in the form of the O-MaSE metamodel, method fragments, and guidelines. In the remainder of the section, we present the three components of the O-MaSE contained in the M1. We first describe the O-MaSE metamodel followed by a description of the method fragments obtained. Finally, we discuss the guidelines that govern the construction of O-MaSE compliant processes.

2 See http://www.fipa.org/activities/methodology.html

3 See http://www.omg.org/cgi-bin/doc?formal/2005-01-06

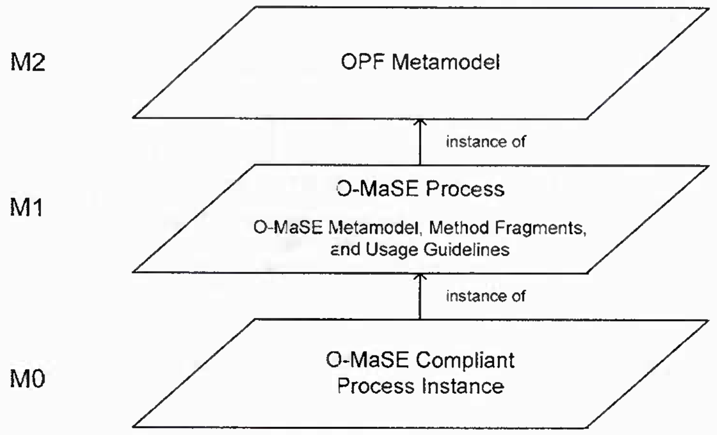

M2
OPF Metamodel
instance of
M1
O-MaSE Process
O-MaSE Metamodel, Method Fragments, and Usage Guidelines
instance of
M0
O-MaSE Compliant Process Instance

Figure 27. O-MaSE Process Framework

3.3.1 Metamodel

The O-MaSE metamodel defines the main concepts we use to define multiagent systems. It encapsulates the rules (grammar) of the notation and depicts those graphically using object-oriented concepts such as classes and relationships (Firesmith & Henderson-Sellers, 2002). The O-MaSE metamodel is based on an organizational approach (DeLoach & Valenzuela, 2007; DeLoach, Oyenan & Matson, 2008). As shown in Figure 28, the *Organization* is composed of five entities: Goals, Roles, Agents, a Domain Model, and Policies. Next we describe each entity in detail.

A *Goal* defines the overall functional of the organization. That is, it could be seen as a desirable situation (Russel & Norvig, 20002) or the objective of a computational process (van Lamsweerde, Darimontm, & Letier, 1998). A *Role* defines a position within an organization whose behavior is expected to *achieve* a particular goal or set of goals. For that reason, each role has specific responsibilities, rights, and relationships defined in order to achieve its overall goal(s). *Agents* are assigned to play those roles and carry out the role's responsibilities using the rights and relationships defined for those roles. In our meta-model, an *Agent* – as described in (Russel & Norvig, 20002) – is an entity that *perceives* and can *perform* actions upon its environment; which includes human as well as artificial (i.e., either hardware or software) entities. In order to perceive and to act on the environment, an agent *possesses* a set of *Capabilities*, which determine how well agents are *capable* of playing roles. Furthermore, roles require a set of capabilities to perform their responsibilities into the organization. These capabilities can be used to capture soft abilities (i.e., plans) or hard abilities (i.e., actions). In fact, *Plans* are associated with access/control over specific resources, the ability to migrate to a new platform, or the ability to use computational algorithms that allow the agent to carry-out specific functional computations. In contrast, *Actions* are associated with hardware of software artifacts (e.g., sensors and effectors) that allow agents to perceive or sense upon a real world environment.

The *Domain Model* captures the real world. The Domain Model maps the environment objects – that include agents – and the relationships between those objects (i.e., environment properties) (DeLoach & Valenzuela, 2007). Likewise, a *Policy constrains* how an organization may behave in a particular situation. In addition, the Domain Model is *used* to define specific organizations policies governing those object types and their relationship with the environment.

Finally, other important concepts mapped in our meta-model are: *Organization Agents* (OA) and *Protocols*. OA are organization that behaves as agents in a higher-level organization. OA captures the idea of organizational hierarchy. In such sense, OA may possess capabilities, may coordinate with other agents, and be assigned to play roles. This concept represents an extension to the traditional Agent-Group-Role (AGR) model (Ferber & Gutknecht, 1998) (Ferber, Gutknetcht, & Michel, 2003) and the organizational meta-model proposed by Odell *et al.* in (Odell, Nodine, & Levy, 2005).

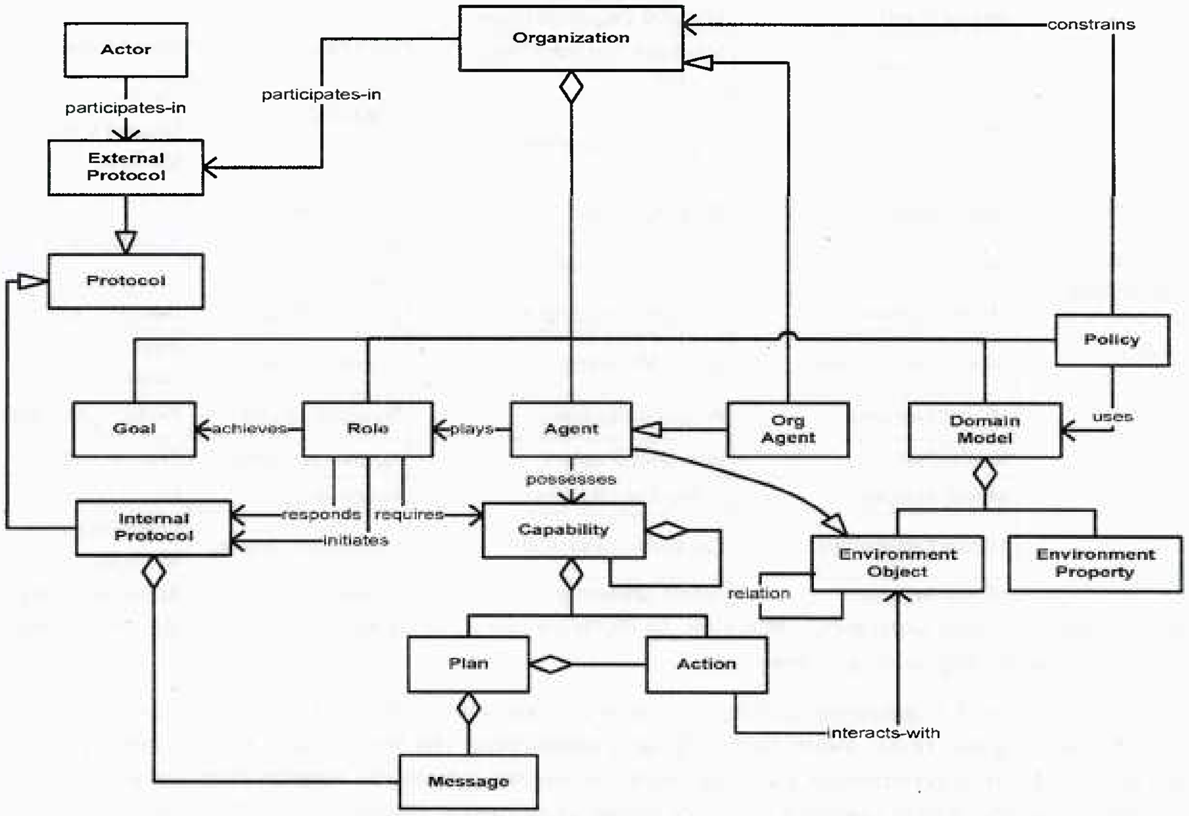


Figure 28. O-MaSE metamodel

The *Protocol* concept helps to capture the idea of interactivity either the organization with external *Actors* or *Agents* and *Roles*. Generally, a protocol "describes a communication pattern s an allowed sequence of messages between agents and the constraints on the content of those messages" (Odell, Parunak, & Bauer, 2001; Odell, Parunak, & Bauer, 2000). A protocol can be of two types: *External Protocol* or *Internal Protocol*. The first maps interactions between the organization and external actors (i.e., humans or other software applications). The second represents a communication pattern between an agent (who initiates) and other (who response) playing a role into the organization.

3.3.2 Method Fragments

As mentioned above, the OPF metamodel defines Stages, Work Units, Work Products, Producers, and Languages, which are used to construct customized processes. In our work, the initial set of method fragments are derived from an extended version of the MaSE methodology (DeLoach, Wood & Sparkman, 2001). O-MaSE assumes an iterative cycle across all phases with the intent that successive iterations will add detail to the models until a complete design is produced. This nicely fits the OPF's Iterative, Incremental, and Parallel Life Cycle model). Our current work focuses on analysis and design. In O-MaSE, we have identified three main activities: (1) requirements engineering, (2) analysis, and (3) design. As shown in Table 7, we decompose each Activity into a set of Tasks and identify a set of Techniques that can be used to accomplish each Task. We also show the different Work Products, Producers, and Languages related to the associated Work Units. Due to the page limitations, we cannot

Table 7. O-MaSE Method Fragments

Work Units				
Activity	Task	Technique	Work Products	Producer
Requirement Engineering, Analysis; and, Design	Model Goals	AND/OR Decomposition	Goal Model	Goal Modeler
	Goal Refinement	Attribute-Precede-Trigger Analysis		
	Model Organizational Interfaces	Organizational Modeling	Organization Model	Organizational Modeler
	Model Roles	Role Modeling	Role Model	Role Modeler
	Define Roles	Role Description	Role Description Document	
	Model Domain	Domain Modeling	Domain Model	Domain Expert
	Model Agent Classes	Agent Modeling	Agent Class Model	Agent Class Modeler
	Model Protocol	Protocol Modeling	Protocol Model	Protocol Modeler
	Model Plan	Plan Specification	Agent Plan Model	Plan Modeler
	Model Policies	Policy Specification	Policy Model	Policy Modeler
	Model Capabilities	Capability Modeling	Capabilities Model	Capabilities Modeler
	Model Actions	Action Modeling	Action Model	Action Modeler

discuss each of these separately[4]. However, to illustrate our basic approach, we describe the details of the requirements engineering activity.

In the Requirement Engineering activity, we seek to translate systems requirement into system level goals by defining two tasks: *Model Goals* and *Goal Refinement*. The first focuses on transforming system requirements into a system-level goal tree while the second refines the relationships and attributes for the goals. The goal tree is captured as a Goal Model as defined in (Miller, 2007). The *Goal Modeler* must be able to: (1) use AND/OR Decomposition and Attribute-Precede-Trigger Analysis (APT) techniques, (2) understand the System Description (SD) or Systems Requirement Specification (SRS), and (3) interact with domain experts and customers. The result of these two tasks is a refined Goal Model.

3.3.3 Guidelines

Guidelines are used to describe how the method fragments can be combined in order to obtain O-MaSE compliant processes. These guidelines are specified in terms of a set of constraints related to Work Units and Work Products, which are specified as Work Unit preconditions and post-conditions. We formally specify these guidelines as a tuple ⟨*Input*, *Output*, *Precondition, Post-condition*⟩ where *Input* is a set of Work Products that may be used in performing a work unit, *Output* is a set of Work Products that may be produced from the Work Unit, *Precondition* specifies valid Work Product/Producer states, and *Post-condition* specifies the Work Product State (see Table 7) that is guaranteed to be true after successfully performing a work unit (if the precondition was true). To formally specify pre and post-conditions, we use first order predicate logic statements defined over the Work Products (WP) and Producers (P), the Work Products states, and the iteration (n) and version (m) of the Work Products. Specific predicates related to work product and producer states are defined in Table 8.

4 A detailed description of the current set of O-MaSE Tasks, Techniques, Work Products, and Producers can be found at http://macr.cis.ksu.edu/O-MaSE/

Figs. 4 – 8 illustrate a set of guidelines for a few of the Tasks defined in Table 7. Figure 29 defines the *Model Goals* task. Inputs to the task may include the *Systems Description* (SD), the *Systems Requirement Specification* (SRS), the *Role Description Document* (RD), or a previous version of the *Goal Model* (GM). Actually, only one of these inputs is required, although as many as are available may be used. The inputs are used by the *Goal Model Producer* (GMP) to identify organization goals. As a result of this task, the Work Product *GM* is obtained.

Table 8. Work Product/Producer States

No.	State	Definition
1	inProcess()	True if the work product is in process.
2	completed()	True if the work product has been finished.
3	exists()	exists() = inProcess() ∨ completed()
4	previousIteration()	True if the work product's iteration is any previous one.
5	available()	True if the producer specified is available to work on the task

Figure 30 depicts the task *Goal Refinement*. Generally, this task only requires as input a GM from the Model Goals task and produces a refined Refined Goal Model model.

Figure 31 shows the task Model Agent Classes, which requires as input a *Refined Goal Model* (RG), an *Organization Model* (OM), or a *Role Model* (RM). As output an *Agent Class Model* (AC) is obtained. In the task, the *Agent Class Modeler* (ACM) identifies the types of agents in the system. A *Capability Model* (CM) may also be used as input because agents may be defined in terms of capabilities. However, the CM is never sufficient or mandatory and thus is termed as an *optional* input (it is not part of the Precondition). The *Protocol Model* (PrM) may be useful in identifying relationships between agents and thus, it is also optional.

The *Model Plan* task is defined in Figure 32. The inputs can include a RG, RM, or an AC, which allow the *Plan Modeler* (PlM) to define plans used by agents to satisfy organization goals. In addition, a PrM, Action Model (AM), and CM are required as input because such plans may require the interaction with other entities using some defined protocol.

Finally, the *Model Protocol* task is defined in Figure 33. To document a PrM, the *Protocol Modeler* (PrP) requires the RM and the AC or a previous iteration of the PrM. The *Domain Model* (DM), OM, and AM are optional inputs to this task; they define actions that the agent may perform on environment objects, which can also be modeled as interactions.

TASK NAME: Model Goals			
Input	Output	Precondition	Postcondition
SD,SRS, RD,GM	GM	((exists(<SD,n,m>) ∨ exists(<SRS,n,m>) ∨ exists(<RD,n,m>) ∨ previousIteration(<GM>)) ∧ available(GMP)	completed(<GM,n,m>)

Figure 29. Model Goal Task Constrains

TASK NAME: Goal Refinement			
Input	Output	Precondition	Postcondition
GM	RG	Completed(<GM,n,m>) ∧ available(GMP)	exists(<RG,n,m>)

Figure 30. Goal Refinement Task Constrains

TASK NAME: Model Agents Classes			
Input	Output	Precondition	Postcondition
RG,RM, OM,AC, CM,PrM	AC	(exists(<RG,n,m>) ∨ exists(<RM,n,m>) ∨ exists(<OM,n,m>) ∨ exists(<SM,n,m>) ∨ previousIteration(<AC>)) ∧ available(ACM)	completed(<AC,n,m>)

Figure 31. Model Agent Classes Task Constrains

TASK NAME: Model Plan			
Input	Output	Precondition	Postcondition
RG,RM, AC,PrM, AM,CM	PlM	((exists(<RG,n,m>) ∧ exists(<AC,n,m>)) ∨ exists(<PrM,n,m>) ∨ exists(<AM,n,m>) ∨ previousIteration(<PlM>)) ∧ available(PlP)	completed(<PlM,n,m>)

Figure 32. Model Plans Task Constrains

TASK NAME: Model Protocol			
Input	Output	Precondition	Postcondition
RM,AC,DM, OM,AM	PrM	((exists(<RM,n,m>) ∧ exists(<AC,n,m>))∨ previousIteration(<PrM>)) ∧ available(PrP)	completed(<PrM,n,m>)

Figure 33. Model Protocol Task Constrains

3.4 WMD Search Example

Next, we present two examples of applying the O-MaSE to derive custom processes. We combine O-MaSE method fragments to create a custom process for a Weapon of Mass Destruction (WMD) system in which agents detect and identify WMD in a given area. There are three types of WMD that can be identified: radioactive, chemical, and biological. Once a suspicious object is found, it must be tested to determine the concentration of radioactivity and nerve agents (chemical and biological). If the object is indeed a WMD, it is removed. The mission is successful when the area has been entirely searched and all the WMD have been removed. In the subsequent subsections, we present two custom processes for the WMD Search application.

3.4.1 Basic O-MaSE Process

The first process we derive is appropriate for a small agent-oriented project in which reactive agents achieve goals that have been assigned at design time. Essentially, the only products required for this type of system are the system goals, agent classes, agent plans, and inter-agent protocols. This type of process leads to a rigid MAS but is very easy and fast to develop. This process may also be suitable for prototyping, where a simple and rapid process is needed.

Figure 34 shows the result of applying O-MaSE guidelines to the creation of our custom process. (Tasks are represented by rounded rectangles while Work Products are represented by rectangles.) The Work Products associated with the products identified above are included, along with the Tasks required to produce them. (We do not show the Producers to simplify the figure, but we assume the appropriate Producers are available.) Connections between Tasks and Work Products are drawn and the preconditions and post-conditions of each Task are verified. Each Task will be discussed below.

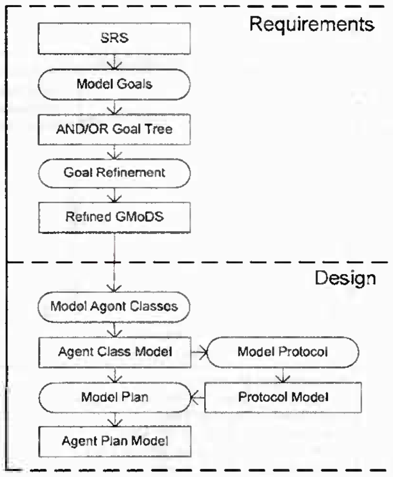


Figure 34. Basic O-MaSE Process

Model Goals/Goal Refinement. From the System Description, the Goal Modeler defines a set of system level goals in the form of an AND/OR goal tree. This initial goal tree is refined into a refined Goal Model as shown in Figure 35. The syntax uses a rectangle with the type designator «Goal». The diamond notation is used to denote AND refined goals (conjunction), whereas the triangle notation is used to denote OR refined goals (disjunction). Refined Goal Models include the notion of goal precedence and goal triggering (Miller 2007). A *precedes* determines which goals must be achieved while a *trigger* relation signifies that a new goal may be instantiated when a specific event occurs during the pursuit of another goal. Figure 35 captures a goal-based view of the system operation.

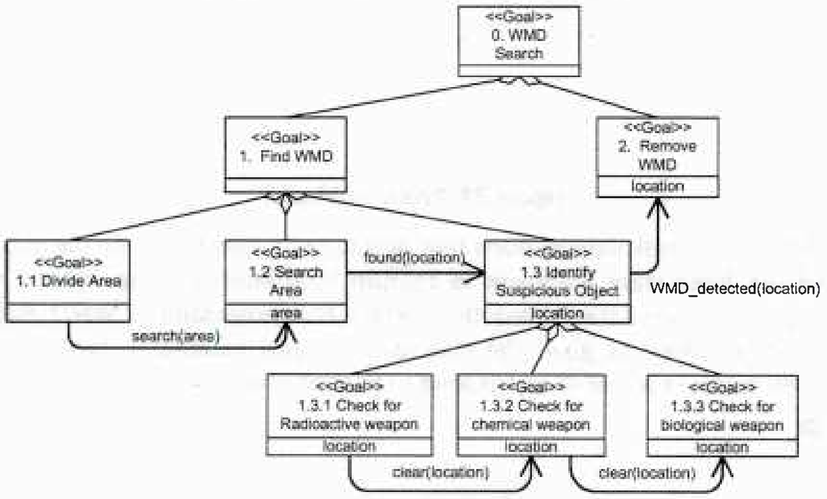
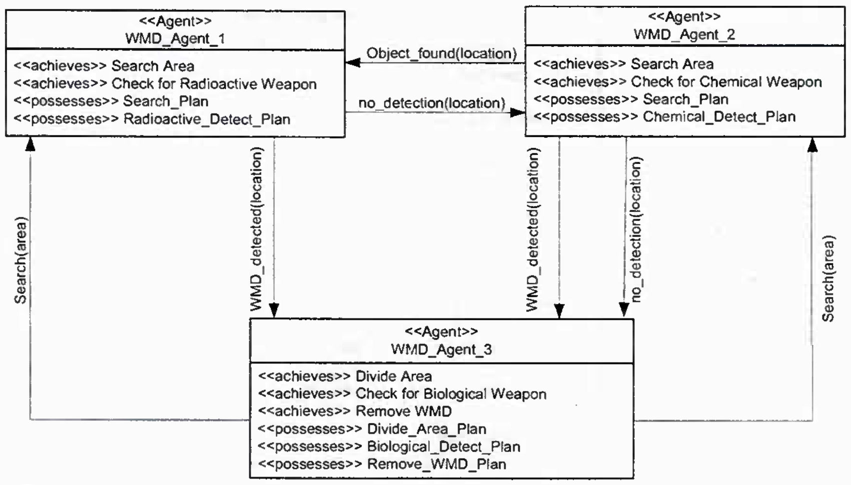


Figure 35. Goal Model

Model Agent Classes. The purpose of this task is to identify the type of agents in the organization and to document them in an Agent Class Model (Figure 36). In our example, agents are defined based on the goals they can achieve and the capabilities they possess as specified by the «achieves» and «possesses» type designators in each agent class (denoted by the «Agent» type designator). Protocols between agent classes are identified by arrows from the initiating agent class to the receiving agent class. The details of these protocols are specified later in the Model Protocols task.



<<Agent>> WMD_Agent_1
<<achieves>> Search Area
<<achieves>> Check for Radioactive Weapon
<<possesses>> Search_Plan
<<possesses>> Radioactive_Detect_Plan
<<Agent>> WMD_Agent_2
<<achieves>> Search Area
<<achieves>> Check for Chemical Weapon
<<possesses>> Search_Plan
<<possesses>> Chemical_Detect_Plan
<<Agent>> WMD_Agent_3
<<achieves>> Divide Area
<<achieves>> Check for Biological Weapon
<<achieves>> Remove WMD
<<possesses>> Divide_Area_Plan
<<possesses>> Biological_Detect_Plan
<<possesses>> Remove_WMD_Plan
Object_found(location)
no_detection(location)
WMD_detected(location)
WMD_detected(location)
no_detection(location)
Search(area)
Search(area)

Figure 36. Agent Class Model

Model Protocol. The Model Protocol task defines the interactions between agents. For example, Figure 37 captures the WMD_detected protocol where WMD_Agent_1, (who is pursuing the Check for Radioactive Weapon goal) detects a WMD and notifies WMD_Agent_3 (who is pursuing the Remove WMD goal). The notification is done by sending a detected message with the location as parameter. Upon reception of this message, an acknowledgment is returned.

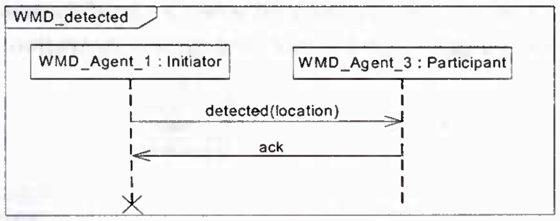


Figure 37. Protocol Model

Model Plan. The Model Plan task defines plans that agents can follow to satisfy the organization's goals. To model this, we use finite state automata to capture both internal behavior and message passing between agents. Figure 38 shows the Radioactive_Detect_Plan possessed by WMD_Agent_2 to achieve the Check for Radioactive Weapon goal. The plan uses the goal parameter, location, as input. Notice that, a plan produced in this task should correspond to all related protocols.

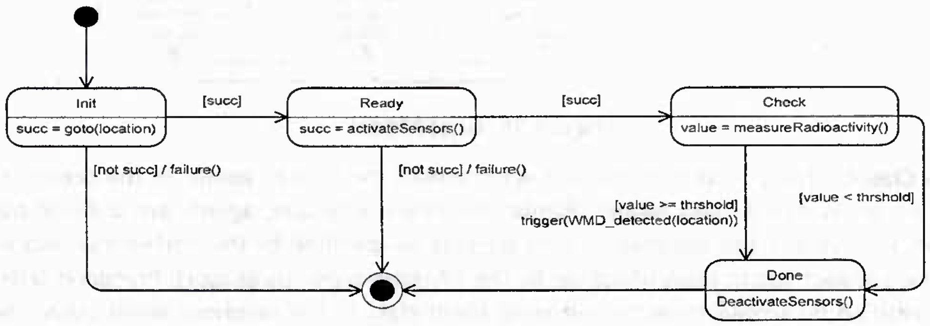


Figure 38. Plan Model

3.4.2 Extended O-MaSE Process

To produce a more robust system that adapts to changes and internal failures, it is necessary to have a process that can produce additional information such as roles and policies. *Roles* define behavior that can be assigned to various agents while *policies* guide and constrain overall system behavior. To accommodate such a system, additional Tasks must be introduced into the process to produce a Role Model and a Policy Model. This type of process allows designers to produce flexible, adaptive, and autonomous systems. Figure 39 shows the custom process for this example. Added tasks are:

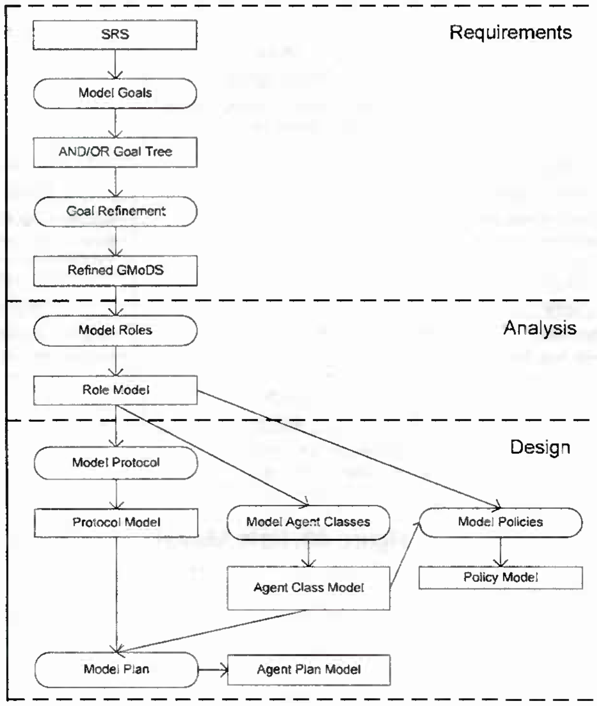


Figure 39. Extended O-MaSE Process

Model Roles. The Model Roles task identifies the roles in the organization and their interactions. Role Modelers focus on defining roles that accomplish one or more goals For example, each role in the Role Model in Figure 40 achieves specific goals from Figure 35. Each role also *requires* specific capabilities.

Model Policy. The Model Policy task defines a set of formally specified rules that describe how an organization may or may not behave in particular situations (Harmon, DeLoach, & Robby, 2007). For example, a policy "An agent may only play one role at a time" can be translated as

$$\forall a1, a2{:}agent,\ r{:}role \mid a1.plays(r1) \wedge a1.plays(r2) \rightarrow r1=r2$$

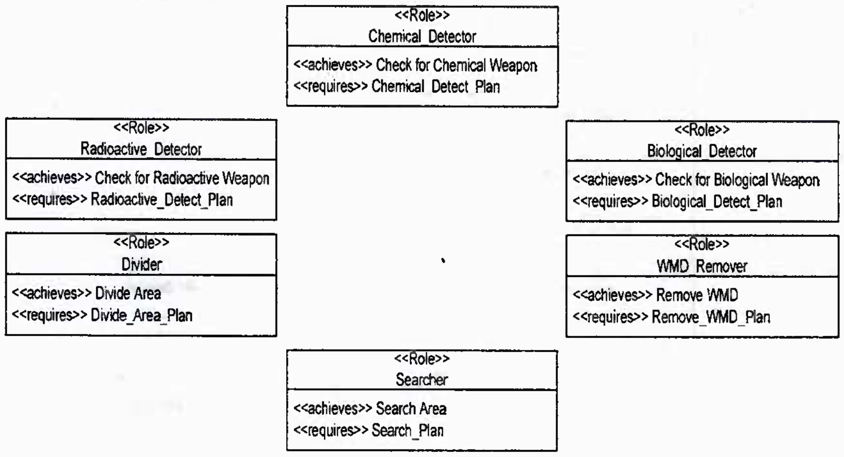
3.5 Conclusions

This section has presented the O-MaSE Process Framework, which allows users to construct custom agent-oriented processes from a set of standard methods fragments. The main advantages of our approach are that:

(1) all O-MaSE fragments are based on a common metamodel that ensures the method fragments can be combined in a coherent fashion,

(2) each method fragment uses only concepts defined in the metamodel to produce work products that can be used as input to other method fragments; and,

(3) the associated guidelines constrain how method fragments may be combined in order to assemble custom O-MaSE compliant processes that produce an appropriate set of products without producing unnecessary products.



<<Role>>
Chemical_Detector
<<achieves>> Check for Chemical Weapon
<<requires>> Chemical_Detect_Plan
<<Role>>
Radioactive_Detector
<<achieves>> Check for Radioactive Weapon
<<requires>> Radioactive_Detect_Plan
<<Role>>
Biological_Detector
<<achieves>> Check for Biological Weapon
<<requires>> Biological_Detect_Plan
<<Role>>
Divider
<<achieves>> Divide Area
<<requires>> Divide_Area_Plan
<<Role>>
WMD_Remover
<<achieves>> Remove WMD
<<requires>> Remove_WMD_Plan
<<Role>>
Searcher
<<achieves>> Search Area
<<requires>> Search_Plan

Figure 40. Role Model

4 THE AGENTTOOL III DEVELOPMENT ENVIRONMENT

The agentTool III (aT^3) development environment is built on the Eclipse platform and provides that traditional model creation tools to support the analysis, design, and implementation of multiagent systems following the Organization-based Multiagent Systems Engineering (O-MaSE) methodology. It also provides verification and metrics computation components. In addition, aT^3 provides the ability to compose, verify, and maintain customized O-MaSE complaint processes.

The agentTool III (aT^3) development environment supports the creation of multiagent systems following the *Organization-based Multiagent Systems Engineering* (O-MaSE) methodology, as discussed in Section 0. aT^3 is a successor to the original agentTool that was developed in 2000 – 2001 at the Air Force Institute of Technology and is currently a project of the Multiagent & Cooperative Robotics Laboratory[5] (MACR) at Kansas State University. aT^3 was developed in Java and built on top of the Eclipse[6] platform and the Eclipse Process Framework[7] (EPF).

The goal of aT^3 is to support and enforce the O-MaSE methodology, which allows users to create their own customized development processes. O-MaSE has three structures – a metamodel, a set of methods fragments, and a set of guidelines – that enable users to use method engineering techniques to create custom processes. The O-MaSE *metamodel* defines the key concepts needed to design and implement multiagent systems while the O-MaSE *method fragments* include the actual tasks that are performed and the work products that are produced. The O-MaSE *guidelines* define how the method fragments are related to one another and thus how customized processes can be built.

aT^3 has five components that are integrated into a single tool. These components are the graphical editor, the process editor, the verification framework, and the code generation facility.

4.1 Graphical Editor

aT^3 supports the graphical editing of models that define all the concepts defined in the O-MaSE's metamodel. aT^3 supports drag-and-drop addition of icons and ensures the only appropriate connections are made between the various kinds of icons. aT^3 also provides pop-up panels for editing the internal detail of the various concepts such as parameters, etc. The Graphical Editor supports the O-MaSE models described in Section 3.3.2.

- ***Goal Model***. This is an AND/OR goal tree structure. Goals in the Goal Tree may have any number of attributes and goals may have precedence and triggering relationships between them. (We may also want to add the notion of ***events*** that ***occur*** during a goal and ***trigger*** other goals.)
- ***Agent Model***. An Agent Class Model defines the agent classes and sub-organizations that will populate the organization. Agent Models are static models that may include agents, actors, organizations, roles, capabilities, and protocols. Relationships between the various entities include: inheritance, possesses, plays, and requires.
- ***Role Model***. A role model is a diagram representing all the roles in the organization along with the goals they are achieving and the interaction protocol existing in the organization. Role

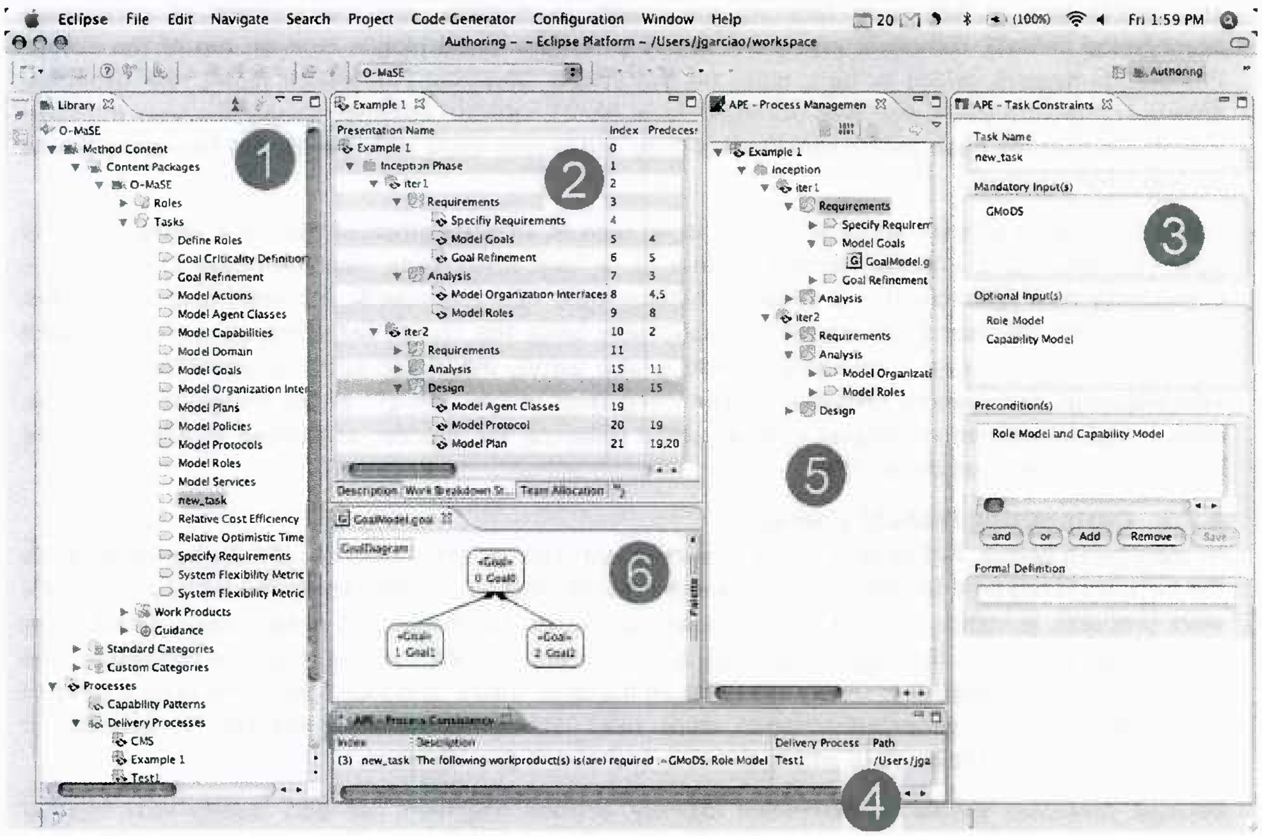
[5] http://macr.cis.ksu.edu/
[6] http://www.eclipse.org/
[7] http://www.eclipse.org/epf/

Models are static models that may include goals, roles, actors, capabilities, and protocols. Relationships between the various entities include inheritance, require, and require.

- ***Organization Model***. The Organizational Model shows the interaction between the organization and the external actors. Organization Models are static models that may include actors, organizations, goals, and protocols. Relationships between organizations and goals are modeled via the *requires* relation.
- ***Protocol Model***. A Protocol Model consists of Protocol Diagrams as specified in AUML. It describes sequences of messages sent between, roles, organizations, and external actors. It also includes alternative and looping structures.
- ***Plan Model***. An agent plan model is based on Finite State Automata. Includes states and transitions. Use Capabilities/actions within states and on transitions.
- ***Capability-Action Model***. PA Capability includes three levels of capabilities: Capabilities, Actions, and Operations. Capabilities may either be (exclusively) composed of a Plan or of other Capabilities/Actions.
- ***Domain Model***. A domain model contains the definition of the environment in which the multiagent system is situated. It consists of a set of Environment Objects (which may include agents), their attributes, and their relationships. The model may also include Environment Policies, which define the processes and principles that govern the multiagent system.
- ***Policy Model***. A policy model is a document containing all the policies applicable to the system.

A screen shot of the aT^3 is shown in Figure 41. On the left side of the screen, the Eclipse Package Explorer allows the user to organize and store O-MaSE models in projects. Generally, subdirectories within projects refer to sub-organizations in the system design, thus the Package Explorer file structure mimics the hierarchical structure of the system. The model shown is an Agent Class Diagram. The icons shown in the Palette on the right side of the screen show the valid components and relations that may be added to the model. To add a component to the model, users simply click on the component in the Palette and then click where they want to place the component in the model. Once the component has been placed in the model, it may be edited or moved to another location. The protocol components are slightly different in that they are added between two actors or agents. To add a protocol, the user first clicks on the protocol icon in the Palette and then on the two actors/agents that participate in the protocol. After placing the protocol, the name may be edited. To add relationships between model components, the user also clicks on the desired relationship in the Palette and then click on two components already in the model. Relationships have fixed names that may not be edited.

The notation used in the aT^3 models is very simple and consistent. Model components are generally represented as a box with several compartments. The top compartment specifies the type of component and the component name. Thus, in Figure 41, agent classes are represented as boxes with an «Agent» type. In the O-MaSE notation, guillemets are used to enclose *type designators* and should not be confused with UML stereotypes. Directly below the type designation is the name of the agent class. Two unique component types are external actors and protocols. External actors are represented using a stick person figure with the name of the actor directly below the figure, while a protocol is represented as an arrow between two components (e.g., roles, agents, organizations, etc.). The name on the arrow is editable and represents the name of the protocol, which can be defined in detail via a protocol model. (A



O-MaSE
- Method Content
  - Content Packages
    - O-MaSE
      - Roles
      - Tasks
        - Define Roles
        - Goal Criticality Definition
        - Goal Refinement
        - Model Actions
        - Model Agent Classes
        - Model Capabilities
        - Model Domain
        - Model Goals
        - Model Organization Inte
        - Model Plans
        - Model Policies
        - Model Protocols
        - Model Roles
        - Model Services
        - new_task
        - Relative Cost Efficiency
        - Relative Optimistic Time
        - Specify Requirements
        - System Flexibility Metric
        - System Flexibility Metric
      - Work Products
      - Guidance
  - Standard Categories
  - Custom Categories
- Processes
  - Capability Patterns
  - Delivery Processes
    - CMS
    - Example 1
    - Test1

Presentation Name	Index	Predeces
Example 1	0	
Inception Phase	1	
iter1	2	
Requirements	3	
Specifiy Requirements	4	
Model Goals	5	4
Goal Refinement	6	5
Analysis	7	3
Model Organization Interfaces	8	4,5
Model Roles	9	8
iter2	10	2
Requirements	11	
Analysis	15	11
Design	18	15
Model Agent Classes	19	
Model Protocol	20	19
Model Plan	21	19,20

- Example 1
  - Inception
    - iter1
      - Requirements
        - Specify Requirem
        - Model Goals
          - GoalModel.g
        - Goal Refinement
      - Analysis
    - iter2
      - Requirements
      - Analysis
        - Model Organizati
        - Model Roles
      - Design

Task Name
new_task
Mandatory Input(s)
GMoDS
Optional Input(s)
Role Model
Capability Model
Precondition(s)
Role Model and Capability Model
Formal Definition

Index	Description	Delivery Process	Path
(3)	new_task The following workproduct(s) is(are) required :-GMoDS, Role Model	Test1	/Users/jga

Figure 42. Screenshot of APE

similar arrow notation is used in the protocol diagram to represent individual messages and in the goal diagram to represent individual events.) Relations between components are represented using traditional object-oriented notation for similar concepts such as inheritance and aggregation, and open headed arrows with fixed labels for O-MaSE specific relationships. In Figure 41, the «plays» arrow is used to define which agents can play which roles, the *possesses* arrow defines which agents possesses which capabilities, and the «requires» arrow is used to define which roles require which capabilities.

As shown in Figure 41, aT^3 also supports embedding of relations to simplify the graphical layout of models. For instance, the *PCmember* agent class has two embedded relations: «plays» *Assigner* and «plays» *Partitioner*. This embedding represents the situation where the model also contains two additional roles and relationships between the *PCmember* and those two roles. In aT^3, the user may toggle between the *embedded mode* and the full mode which shows all relations explicitly.

4.2 Process Editor (APE)

The agentTool Process Editor (APE), a tool for creating and customizing processes for multi-agent system development. APE is an Eclipse-based[8] plug-in which uses EPF to facilitate the management of tailored agent-based processes based upon the O-MaSE Process Framework (hereafter, simply referred to as O-MaSE) (Garcia-Ojeda, DeLoach, & Robby, 2008). O-MaSE provides the concepts, rules, methods fragments, and guidelines needed by process engineers to assemble O-MaSE compliant processes, while APE provides the infrastructure to develop, verify, and maintain O-MaSE compliant processes.

[8] http://www.eclipse.org/

The overall goal of APE is to facilitate the design, verification, and management of customized agent-based O-MaSE compliant processes. APE is an Eclipse-based plug-in built on top of the Eclipse Process Framework, which in turn, relies on the Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF), the Web Standard Tools (WST), and the Common Properties User Interface (UI). It also uses the open source components JTidy and Lucene along with specific EPF tool components such as UMA services[9].

Figure 42 shows a screenshot of APE. APE can be seen as an aggregation of five basic views: a Library view (Window 1), a Process Editor (Window 2), a Task Constraints view (Window 3), a Process Consistency view (Window 4), and a Process Management view (Window 5). It is important to note that *Window 1* and *Window 2* use base EPF functionality while the rest of the windows were developed specifically for APE as an Eclipse plug-in. APE is also loosely integrated with the agentTool III[10] (aT^3) –development environment (Window 6). The integration of APE and aT^3 allows designers to analyze, design, and implement multiagent systems while following the process as defined in APE. Next, we introduce the main features of APE.

4.2.1 Customizing O-MaSE Content

As mentioned before, APE directly uses EPF components such as EPF Authoring. EPF Authoring provides the basic tools required for process engineers to manage method content elements such as roles, tasks, work products, guidance, content categories and processes. For instance, if a development team may need to perform tasks that are not defined in the current O-MaSE method fragment repository (see Window 1), process engineers simply right-click on the label "Tasks" and then specify the main attributes of the new tasks (i.e., general description, steps, roles, work products, and guidance). For example, in Window 1, we show the result of adding the *new_task* task in the repository.

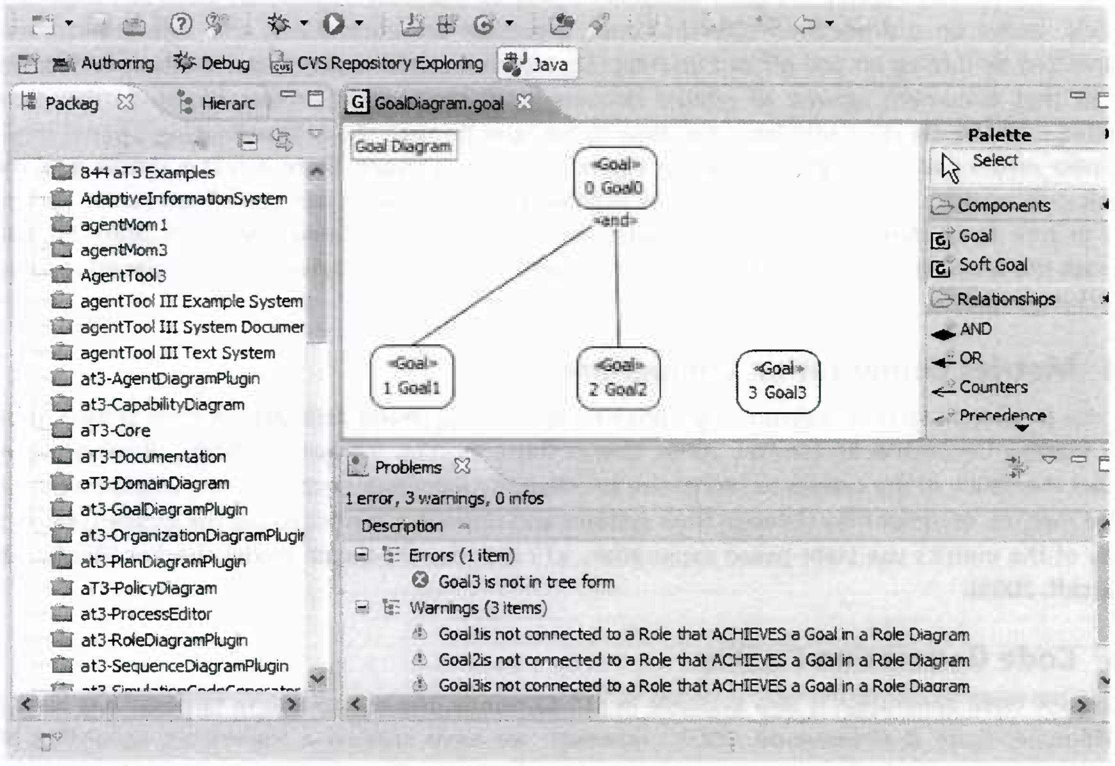
Because processes should be assembled logically, process engineers can also specify how method fragments are related to one another by adding/modifying task pre- and post-conditions. To modify these conditions, the process engineer selects a task from Window 1 and modifies an existing condition or specifies new conditions in Window 3. Preconditions are specified by selecting work products from the *Optional Inputs(s)* list in Window 3, and including them in the logical equation (e.g., (*work_product_1* ∧ *work_product_2)* ∨ *work_product_3*). APE automatically verifies the precondition before it is added into the *Precondition(s)* list. This verification was developed using Another Tool for Language Recognition[11] (ANTRL). Window 3 shows the precondition (i.e., *Role Model and Capability Model*) specified for the *new_task* task.

The purpose of pre- and post-conditions is to help process engineers in the process of verifying the consistency of their customized processes. APE also provides a mechanism for automatically verifying customized processes (Window 2). Any problems found during this verification steps are displayed as processing inconsistencies in Window 4. In this case, adding the new_task task to the Test1 process (not shown) results in an error since the *new_task* task was inserted before the *Goal Refinement* task, resulting in the inconsistency shown in Window 4.

4.2.2 Engineering Custom Processes

In Window 2, the software development team – perhaps supervised by a process engineer and/or a project manager – has the challenge of creating a "tailored" or "customized" process. Such processes define sequences of tasks, which are performed by roles that produce specified work products. These

[9] http://www.eclipse.org/epf/composer_architecture/
[10] http://agenttool.cis.ksu.edu/
[11] http://antlr.org/



1 error, 3 warnings, 0 infos

Errors (1 item)
Goal3 is not in tree form
Warnings (3 items)
Goal1is not connected to a Role that ACHIEVES a Goal in a Role Diagram
Goal2is not connected to a Role that ACHIEVES a Goal in a Role Diagram
Goal3is not connected to a Role that ACHIEVES a Goal in a Role Diagram

Figure 43. Verification Framework

processes are used to describe the desired lifecycle for a specific project (e.g., waterfall, incremental, iterative, etc.).

To create a new custom process in APE, process engineers must select the "Process" label, then right-click on the "Delivery Processes" label (see Window 1), and finally, specify a name for the process and choose O-MaSE as default method library. Once the new process has been created, the process designer can add children (phases, iterations, activities, tasks, or milestones) to it in Window 2. Children are added by right clicking on the process (or other child) name in Window 2 and then selecting a phase, iteration, or task from the method library.

4.2.3 Managing Custom Processes

To help manage custom processes, APE provides three basic functionalities via the Process Management view shown in Window 5. First, it provides the function for uploading documentation regarding the requirements for the given project. Second, it integrates aT^3, so managers and developers can directly view and edit the work products expected for each task. Finally, APE also provides support for entering and tracking budget and scheduling information using Earned Value Analysis (EVA) (Fleming & Koppelman, 2006). The core concept of EVA is *earned value*, which refers to the cost of work performed at a given point according to a process development plan (American National Standards Institute, 1998). Thus, the foundation of the EVA is a good work breakdown structure with clearly defined tasks (see Window 2), each of which has been assigned a cost and a task completion date.

4.3 Verification Framework

The aT^3 Verification Framework gives designers a way to maintain consistency between their O-MaSE models, based on a predefined rule set. Since processes are customized, this rule set can also be customized by turning on and off certain rules. Each time a model is saved, the Verification Framework checks that document against all related documents in the current project based on the currently enabled rules. Verification problems are show to the user through the Eclipse Problems panel similar to compiler errors and warnings as shown in Figure 43. In this project, there is just a simple goal model, which is incorrectly drawn. As shown in the Problems panel, Goal3 is not connected to the rest of the goal in tree form. Also, since the role model has not yet been defined, the Verification Framework displays the warning that none of the leaf goals (Goal1, Goal2, or Goal3) have been assigned to any roles for achievement.

4.4 Metrics Computation Component

aT^3 also provides a suite of metric computation tools that help developers analyze their system at design time (Robby, DeLoach & Kolesnikov, 2006). One of those metrics, System Flexibility, allows designers to predict the ability of the system to reorganize to overcome individual agent failures. Based on the results of the metrics, designer may redesign their systems and rerun the metrics to get the desired result. Since many of the metrics use state-based exploration, aT3 includes the Bogor model checker (Robby, Dwyer &Hatcliff, 2003).

4.5 Code Generation Facility

Automatic code generation is also available in aT^3. Currently, the only platform targeted has been JADE (Bellifemine, Caire & Greenwood, 2007). However, we have created a framework consisting of the Organization, Operation, Social, and Environment levels. At the Organization level, agents and roles are chosen for achieving specific goals. At the Operation level, agents achieve goals by performing actions based on their available capabilities. At the Social level, agent's interactions are captured via messaging, while at the Environment level, the knowledge of object types and relationships are generated. Due to the detail of the O-MaSE models, the aT^3 JADE generator is capable of generating 100% of the code necessary to create functional JADE systems.

4.6 Availability

APE is packaged with aT^3 as an Eclipse plug-in and is available for download from the aT^3 website[12]. The website includes download and installation procedures, as well as a complete set of online tutorials for most aT^3 and APE functions.

[12] http://agenttool.cis.ksu.edu/

5 Predictive Metrics for Multiagent Systems

Designing correct models for multiagent systems is a tedious task. This task becomes vastly more difficult as the complexity of the system increases. The designer has to take the requirements of the system and design a system that is flexible enough to handle failure, but efficient enough to accomplish the task in an acceptable time frame. The ability to adapt to failure can be a principle reason for using a multiagent system. However, it is this inherent flexibility of multiagent systems can lead to unwanted emergent behavior. The goal in this part of the research project was to define a set of metrics that could be used at design time to predict the behavior of the multiagent system in order to restrict unwanted behavior.

Our goal was to use design knowledge required for OMACS-based systems in order to predict the behavior of the system. Specifically, we focused on four key OMACS concepts for this investigation: goals, roles, agents and capabilities. The goal model provides an abstract representation of the state of the system with temporal constraints. As the number of goals, roles, agents and capabilities increase, the number of interactions can exponentially increase. This makes human analysis very tedious and time consuming as the organization size increases. Therefore, automation is needed in the analysis of these models and how they interact with one another.

5.1 Capturing Traces for Metric Computation

Metrics are a powerful tool that can be used in all phases of software development. The design of a metric suite requires the definition of a framework for metric integration, which makes it easier to write new metrics. The framework allows users and experts to define metrics for use it the development process. Useful metrics can then be shared with other developers in the multiagent field. Additionally our metrics provides the baseline that allows us to apply abstractions to the system. These abstractions can leverage the ideas from abstract interpretation in order to create much more efficient methods of proving properties of our models.

Due to the use of the GMoDS goal model, which allows the triggering of new goal instances based on events that occur during system operation, an OMACS-based system has a possibly infinite set of goals. This possibly infinite number of goals poses a problem for model checking systems, as the state space of the system will be infinite and the model checker will never terminate. As the number of events in the system increases, the time to model check will increase exponentially. To make the possible number of goals in the system finite, we provide the user with the ability to specify *event bounds* (specified in the form of a range, m .. n where $0 \leq m \leq n$) that limit the number of times a specific event may occur.

5.1.1 Trace Types

There are three types of system traces that we are interested in: Goals (**G**), Goals-Roles (**GR**), and Goals-Roles-Agents (**GRA**). The **G** traces reveal how many ways there are to achieve the top-level goal without regard for their assignment to roles and agents. Adding the roles to the traces (resulting in **GR** traces) includes the assignment of the various roles that can be used for the achievement of each goal. Finally, the addition of the agents in the traces (yielding **GRA** traces) shows which specific agent was playing the role for when the goal was achieved.

5.1.2 System Traces

As described in Section 1.3.2, many event types may occur and be observed by the system. In this work, we limit the events of interest to those specified in Table 9. For this work, we define a *system trace* as a finite sequence of these such events e_0 … e_n that leads to the achievement of the top level goal ($C(g_0)$). The set of all unique traces is denoted as T_{SUCC}.

Table 9. Events and Properties of Interest

System Events	Definition
$C(g_i)$	goal g_i has been completed (achieved)
$F(g_i)$	goal g_i has been completed (failed)
$T(g_i)$	goal g_i has been triggered.
$A(a_i, r_j, g_k)$	agent a_i has been assigned role r_j to achieve goal g_k.

In the traces, we actually capture *system events* as tuples, *<EventNumber, Event>*, where the *EventNumber* is an Integer and it denotes the time when the event happened and the *Event* is the event that happened. The event must be a *goal achievement* ($C(g_i)$) or a *goal trigger* ($T(g_i)$).

5.1.3 Generating System Traces via Model Checking

We use the Bogor model checker to generate system traces. In general, the state of the system includes the state of all the goal instances in the system; each goal is typically either preceded, active, achieved, or obviated. In each system state, Bogor can take one of two actions: create new goal instances or achieve existing goals. New goal instances are created via the triggers relation. Thus, if a goal has been assigned to an agent and it has a trigger relation from it to a second goal, a new instance of the second goal may be created if the upper bound on the trigger has not been reached. Trigger relations have lower and upper bounds that determine the minimum and maximum number of times that the trigger must/can be executed. If the lower bound has not been reached, then the minimum number of triggers is automatically executed, resulting in possibly several new goal instances. If the maximum has not been reached, then the trigger *may* be executed resulting in the creation of a new goal instance. To allow backtracking, we also keep track of which goals were added to reach the current state, the state of the goal instances in the previous state, and the event that caused the state transition. After each transition, Bogor checks to see if the top-level goal of the system has been achieved. If the top-level goal has been achieved, then no more actions are executed and the current trace is output.

5.1.3.1 The Effect of Policies on System Traces

The effect of arbitrary system policies can have a dramatic effect on system traces. These policies are a problem when the model checker is matching states. To illustrate the problems that can occur, we present a simple example. Assume we have a simple goal model with a single parent goal A, which is AND-decomposed into three sub-goals: B, C, and D. In addition, assume we have the policy

$$C(B) \wedge A(a_1,r_1,C) \Rightarrow \neg A(a_2,r_2,D)$$

The policy states that if goal C has been achieved and goal B has been assigned to an agent, then the system cannot assign goal D. The problem with a policy such as this is that when model checking, the output of the model checker is dependent on the path the system took to get to the current state. If the model checker achieves goal D, then goal A is achievable, but if the system achieves B first then goal D is unachievable. To handle this type of situation, we produce two types of output. In the first case, we assume order of goal achievement is not important and the output consists of the set of all possible goal sets that achieve the top-level goal. As the output is a set based, it loses information about the order of goal achievement. However, the output is considerably smaller and metric computation can be much faster. In the second case, the output is in the form of a tree that captures each state and the transitions between the states.

Table 10. Achievement Traces

Achieved	Assigned
{ }	{A, B, C, D}
{C}	{A, B, D}
{B, C}	{A, D}
{A, B, C, D}	{ }
{B}	{A, C, D}
{B, C}	{A, D}
{A, B, C, D}	{ }

This output, while much larger, retains the information related to the ordering of events and goal achievements. If we have a system with policies, we can run the policies against the second output and prune traces that violate a policy.

5.1.3.2 Optimizing Trace Generation

The efficiency of our trace generation algorithm needs to be reasonable. If the trace generation process takes too long, the results will be of little benefit. The original algorithm as described above was slow, about 80 transitions per second.

Therefore, we started looking for ways to optimize the system. The results are two optimizations that lead to an order of magnitude increase in efficiency. After implementing the optimization described below, we increased our trace generation speed from 80 to 600 transitions per second. We made two significant optimizations: (1) not allow the system to generate traces with unnecessary or redundant goal achievements, and (2) abstracting away the ordering of the events that created goals.

5.1.3.2.1 Unnecessary Achievements

An OR-decomposed goal becomes achieved as soon as one of its sub-goals is achieved. Thus, we can use this definition to keep from generating traces in which more than one of the children of an OR-decomposed (disjunctive) goal are achieved. Once a sub-goal of an OR-decomposed goal has been achieved, we say that all other children of the OR-decomposed goal are *obviated*, and thus not assigned for achievement. We optimize the traces by not including obviated goals when we are generating the state identifier.

5.1.3.3 Symmetric Instance Tree Matching

The second optimization was required to simplify the checking of symmetric goal instance trees. *Symmetric goal instance trees* are defined as two instance trees that are identical except for the parameter values of the goal instances in the tree. This is important to reduce the amount of model checking done. During system operation, while goals are created via a sequence of event triggers, when generating traces, we do not keep track of the order of events that brought us to the current state of the goal instance tree. For example, Figure 44 and Figure 45 show two symmetric goal instance trees. In the first tree, event A occurred followed by event B while in the second tree, the order of the events was reversed. In both trees, one of the *Sweep Area* goals has been achieved. Thus, disregarding the parameter values, we see that these trees are identical.

The method used to identify this situation involves a bottom up generation of unique identifiers for elements of interest in the current state. The *elements of interest* are related to the type of trace we are generating: **G**, **GR**, or **GRA**. **G** traces only require goal instance for the elements, while the **GR** combines the goal and role, and the **GRA** combines the goal, role, and agent. When an element is encountered, the lookup table is checked to see if a unique identifier has been defined for it. If so, the stored unique identifier is returned. If not, a new unique identifier is returned and the element value and unique identifier pair are added to the lookup table.

The element value consists of the concatenation of information related to a goal instance, which includes goal, role, agent, and the state of the goal. The GMoDS execution model defines that each goal instance is in exactly one state at a time: preceded, active, achieved, or obviated. For example, the unique identifiers for each element in Figure 44 and Figure 45 are shown Table 11. For each element containing a leaf goal, the element value (the concatenation of the goal, role, agent, and goal set) is serialized and used to look up the unique identifier. For example the *Mop Area(A)* and *Mop Area(B)* goal both map to *Mop Area*, *Active*, which has the unique identifier 1. For each element containing a parent goal, a number pair related to each of its children is appended. The number pair consists of two values:

number of matching children and the unique identifier, (number of matching elements, unique identifier). For example, for the *Clean Tile (B)* goal in Figure 44, the initial element value is *Clean Tile, Active*; however, since it has two children, the unique identifiers of each of its children are appended resulting in the element value of *Clean Tile, Active, (1,1) (1,3)*. Thus, the goal has two children, the first has once instance and the unique identifier of 1, while the second also has one instance with a unique identifier of 3.

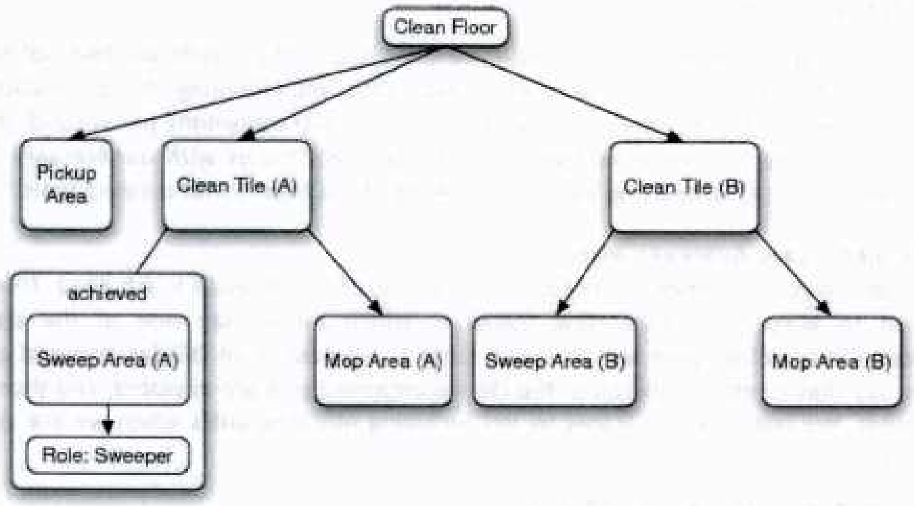


Figure 44. Instance Tree A

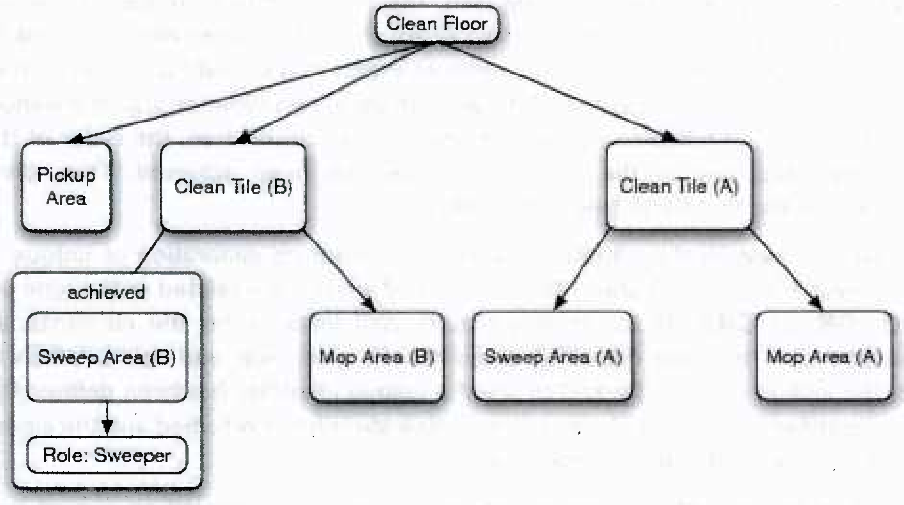


Figure 45. Instance Tree B

Our approach provides us several properties that we can leverage for increased efficiency. First, multiple instances of a goal map to the same unique identifier. Second, different instances of the same child goal - if different in role, agent, or state – map to different unique identifiers. Finally, state comparison can now be done in constant time. And, since state comparison is performed at every transition, performing it in constant time makes a huge impact in model checker efficiency.

Table 11 . State Identification Table.

Unique Identifier	Element Value
0	Pickup Area, Active
1	Sweep Area: Sweeper, Achieved
2	Sweep Area, Active
3	Mop Area, Active
4	Clean Tile, Active, (1,1) (1,3)
5	Clean Tile, Active, (1,2) (1,3)
6	Clean Floor, Active, (1,4) (1,5) (1,0)

5.1.4 Adding Goal Instantiation via Triggering

A shortcoming of our initial work on our flexibility metric (Robby, DeLoach & Kolesnikov, 2006), was the lack of support for the creation of new instance goals via goal triggering. However, goal triggering was a critical part of the goal execution model developed for GMoDS-based systems. To incorporate goal triggering, we decided to use the existing implementation of the GMoDS execution model, which we use to track the current state of the system goals at runtime. The benefits of using our existing execution model implementation was that it

(1) already included goal triggering,
(2) ensured that the goal model state in our model checking would correspond exactly with the goal model state in implemented systems, and
(3) is supported by the agentTool development environment for designing GMoDS goal models that could be incorporated directly into the GMoDS execution environment.

However, the GMoDS execution model had one drawback for use in model checking; it simply maintained the current state of the goal model, which did not allow us to perform typical model checking operations such as backtracking and state comparison. Therefore, we extended the GMoDS execution model to include the ability to

(1) backtrack from the current state
(2) store incremental changes in the state
(3) quickly compare states (with symmetric abstraction)
(4) enumerate events (achievement and triggers)
(5) select events to pursue

5.2 The Occurrence Metric

The definition of T_{Succ} allows us to measure how often a specific design element (here a goal, a role, or an agent) was used in a successful achievement of the top-level goal. We can therefore infer information about a specific element by counting the number of traces in T_{Succ} in which it occurred. Thus, for instance, we define the *Occurrence*, O, of a design element *e* is defined as

$$O(e) = \frac{|\{t \mid t \in T_{SUCC} \wedge t.contains(e)\}|}{|T_{SUCC}|} \qquad (18)$$

The objective of counting occurrences is to identify the design elements that lie on the extremes of the occurrence spectrum. That is, the design elements that occur in almost all successful traces as well as those that occur in very few, or none, of the successful traces. The goal is to identify design flaws or inefficiencies that, if addressed early in development, can improve system performance with little additional cost.

5.2.1 Goal Occurrence Extremes

The two extreme cases of design elements occurrences are very interesting. In the first situation, a design element occurs in the maximum (or almost maximum) number of traces. Thus O(e) = 1, or very close to it. In the second situation of interest, the design elements is used in a minimal number (or very few) successful traces. This indicates that the design element is seldom required in successful traces. Each of these situations is discussed in more detail below.

5.2.1.1 Maximal Traces

Design elements that appear in the maximal number of traces are the most needed and the least restricted design elements of the system. When these design elements are not achievable, then it may no longer be feasible for the system to achieve the goal of the system. The system is definitely not feasible when the design element is needed in 100% of the traces and there is a failure related to that design element.

5.2.2 Floor Cleaning Example

An example multiagent system is the Cooperative Robotic Floor Cleaning Company (CRFCC) as described in Section 2.3.2. Slightly modified versions of the Goal, Role, and Agent Models are shown in Figure 46 – Figure 48. The number of times each goal occurs based on the **G** set of traces is shown in Table 12. As we can see *DivideArea* and *PickupArea* are used in the maximal number of traces, and thus if these goals cannot be achieved, our entire system will fail. Goals that appear in a minimal number of traces are generally restricted by policies, precedence, or triggering. They will only occur when all of the restrictions are met, or the right set of events happen. If the system designer expects these goals to occur in more traces, it may indicate a design flaw that the designer should address.

5.2.2.1 Goal, Role, and Agent Occurrence

Next, we include the Role Model (Figure 48) by looking at the **GR** set of traces, the results of which are shown in Table 13. Notice that six different roles that are defined in the model and in general, the introduction of roles can cause the goal occurrence of any particular goal to increase or decrease. In this example, each goal that occurred in the **G** traces also occurred in the **GR** traces. Thus, we know that each goal is achievable by some role in the model. We can also see that each role occurs in the traces, which shows that each role satisfies some goal in the model.

The final type of model to incorporate into the traces is the Agent Model, resulting in the **GRA** set of traces. Figure 48 shows a system design with five different types of agents. Each agent type possesses a set of capabilities, making them capable of the roles required to achieve the system goals.

The occurrences of the various goals, roles, and agents from the **GRA** traces are shown in Table 14. We can see that *Vacuum Area* goal, the *Vacuumer* role and *Agent 5* do not occur in any traces. If we carefully inspect the Role and Agent Models, we can see that the *Vacuum Area* goal requires the *Vacuumer* role, and the *Vacuumer* role requires the *Vacuum* and *Sweep* capabilities; however, we have no agents that possess both of those capabilities. The design error is in this entire chain of dependencies between the goals, roles and agents. The best way to correct the problem is application dependent. For example, it may be the case that the goals and roles may not be changed leaving only the Agent Model to be modified. Therefore, we can modify the Agent model to allow an agent type to be capable of playing the *Vacuumer* role. In our new Agent Model as shown in Figure 49, *Agent 5* has been given the additional capabilities of *Sweep* and *Move*. The results of this design change are shown in Table 15 where we can see that the *Vacuum Area* goal, the *Vacuumer* role and *Agent 5* occur in traces of the modified system. Notice also that the total number of traces (ways to achieve the top level goal) has increased.

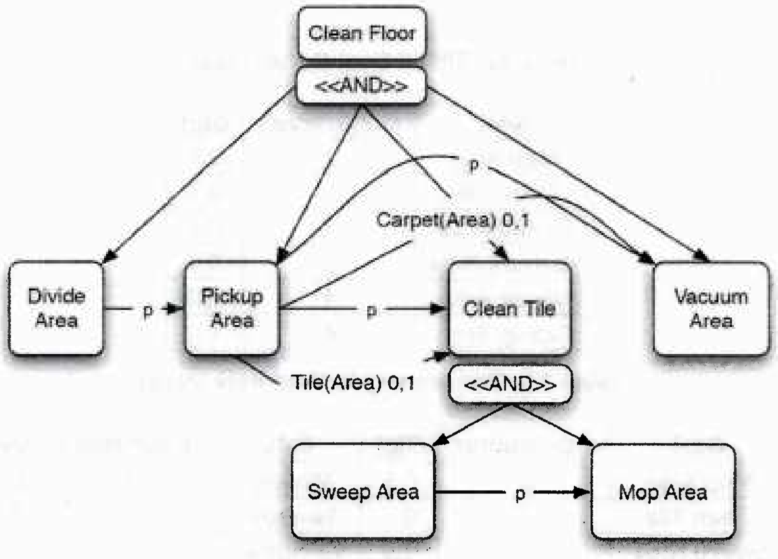


Figure 46 . CRFCC Goal Model A

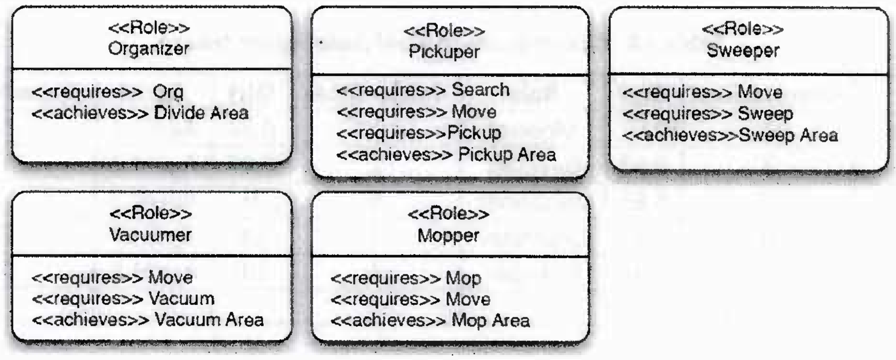


Figure 47 . CRFCC Role Model A

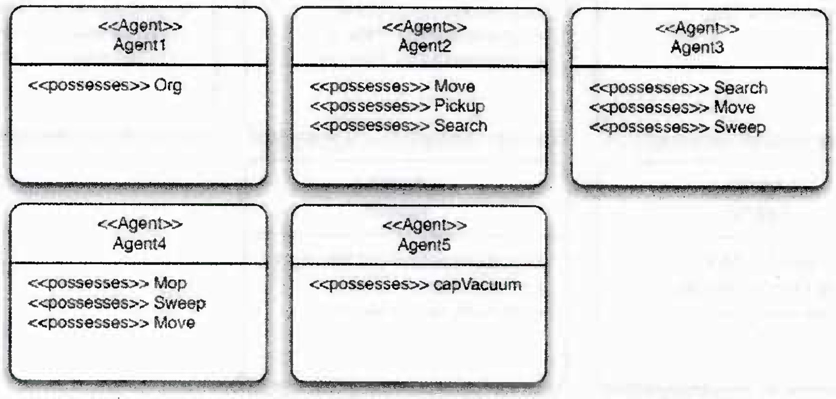


Figure 48 . CRFCC Agent Model A

Table 12. CRFCC Goal Occurrences

Goal	Occurrences	O(g)
Mop Area	2	0.5
Clean Tile	2	0.5
Sweep Area	2	0.5
Vacuum Area	2	0.5
Divide Area	4	1.0
Pickup Area	4	1.0

Table 13 . Occurrences in Goal-Role traces

Goal	Occurrences	O(g)	Role	Occurrences	O(r)
Mop Area	2	0.5	Mopper	2	0.5
Clean Tile	2	0.5	Sweeper	2	0.5
Sweep Area	2	0.5	Vacuumer	2	0.5
Vacuum Area	2	0.5	Organizer	4	1.0
Divide Area	4	1.0	Pickuper	4	1.0
Pickup Area	4	1.0			

Table 14 . Occurrences in Goal-Role-Agent traces

Goal	Occurrences	O(g)	Role	Occurrences	O(r)	Agent	Occurrences	O(a)
Mop Area	2	0.67	Mopper	2	0.67	Agent 1	3	1.0
Clean Tile	2	0.67	Sweeper	2	0.67	Agent 2	3	1.0
Sweep Area	2	0.67	Vacuumer	0	0	Agent 3	1	0
Vacuum Area	0	0	Organizer	3	1.0	Agent 4	2	0.67
Divide Area	3	1.0	Pickuper	3	1.0	Agent 5	0	
Pickup Area	3	1.0						

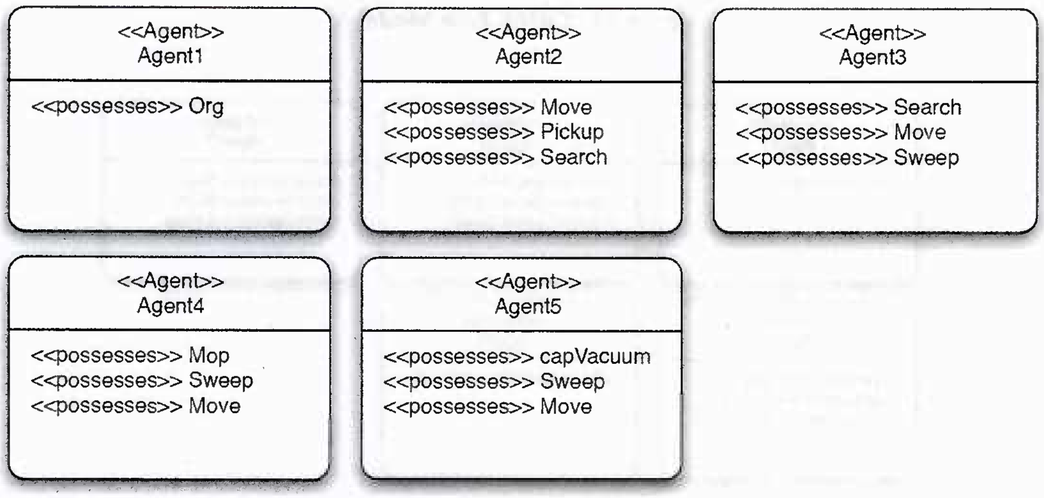


Figure 49. CRFCC Agent Model B

Table 15 . Occurrences using updated Agent Model

Goal	Occurrences	O(g)	Role	Occurrences	O(r)	Agent	Occurrences	O(a)
Mop Area	6	0.67	Mopper	6	0.67	Agent 1	8	1.0
Clean Tile	6	0.67	Sweeper	6	0.67	Agent 2	8	1.0
Sweep Area	6	0.67	Vacuumer	4	0.5	Agent 3	2	0.25
Vacuum Area	4	0.5	Organizer	8	1.0	Agent 4	6	0.67
Divide Area	8	1.0	Pickuper	8	1.0	Agent 5	5	0.63
Pickup Area	8	1.0						

5.2.2.2 Maximal traces

Table 15 also shows that several goals, roles and agents are in the maximal number of traces. For example, the *Divide Area* and *Pickup Area* goals are in all of the traces. Therefore, if either of these goals cannot be achieved then the system cannot achieve the top-level goal. Therefore, it would behoove the designer to focus special attention on these areas as a failure there would surely cause total system failure.

6 References

Abdallah, S., & Lesser, V. (2007). Multiagent Reinforcement Learning and Self-Organization in a Network of Agents. In *Proceedings of the sixth internationol joint conference on outonomous ogents ond multi-ogent systems* (p. 172-179). Honolulu: IFAAMAS.

American National Standards Institute. Earned Value Management Systems. Standard ANSI/EIA-748-A-1998, American National Standards Institute (ANSI), New York, NY, 1998.

Artikis, A., Sergot, M., & Pitt, J. (2007). Specifying norm-governed computational societies. *ACM Tronsoctions on Computotionol Logic.*

Bellifemine, F.L., Caire, G., & Greenwood, D. Developing Multi-Agent Systems with JADE. Wiley & Sons, England, 2007.

Bergenti F., Gleizes M.-P., and Zambonelli F. (eds.): Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Kluwer Academic Publishers (2004).

Bernon C., Cossentino M., and Pavón J.: Agent Oriented Software Engineering. *The Knowledge Engineering Review*. 20(2005) 99–116.

Bernon C., Cossentino M., Gleizes M., Turci P., and Zambonelli F.: A study of some multi-agent meta-models. In: Odell, J., Giorgini, P., and Müller, J. (eds.): *Agent Oriented Softwore Engineering V*. LNCS Vol. 3382. Springer-Verlag, Berlin Heidelberg New York (2004) 62–77.

Bernon, C., Gleizes, M., & Picard, G. (2005). Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. *Agent-oriented Methodologies.*

Bernstein, D., Givan, R., Immerman, N., & Zilberstein, S. (2002). The Complexity of Decentralized Control of Markov Decision Processes. *Mathemotics of Operations Reseorch*, *27*(4), 819–840.

Beydoun G., Gonzalez-Perez C., Henderson-Sellers B., Low G.: Developing and Evaluating a Generic Metamodel for MAS Work Products. In: Garcia, A., Choren, R., Lucena, C. Giorgini, P., Holvoet, T., and Romanosky, A. (eds.): *Softwore Engineering for Multi-Agent Systems IV*. LNCS Vol. 3194. Springer-Verlag, Berlin Heideberg New York (2005) 126–142.

Bradshaw, J., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M., Acquisti, A., Benyo, B., Breedy, M., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., & Hoof, R. V. (2003). Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. In *AAMAS '03: Proceedings of the second internationol joint conference on outonomous ogents and multiogent systems* (pp. 835–842). New York, NY, USA: ACM Press.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., & Perini, A. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 203–236.

Brinkkemper, S.: Method Engineering: Engineering of Information Systems Development Methods and Tools. *Journol of Informotion ond Softwore Technology*. 38(4) (1996) 275–280.

Büchi, J. R. (1960). On a decision method in restricted second-order arithmetics. In *Proceedings of internotionol congress of logic methodology ond philosophy of science* (pp. 1–12). Palo Alto, CA, USA: Stanford University Press.

Bulka, B., Gaston, M., & desJardins, M. (2007). Local strategy learning in networked multi-agent team formation. *Autonomous Agents ond Multi-Agent Systems*, *15*(1), 29–45.

Chaki, S., Clarke, E. M., Ouaknine, J., Sharygina, N., & Sinha, N. (2004). State/event-based software model checking. In E. A. Boiten, J. Derrick, & G. Smith (Eds.), *Proceedings of the 4th internotionol conference on integroted formol methods (ifm '04)* (Vol. 2999, pp. 128–147). Springer-Verlag.

Chiang, C., Levin, G., Gottlieb, Y., Chadha, R., Li, S., Poylisher, A., Newman, S., Lo, R., & Technologies, T. (2007). On Automated Policy Generation for Mobile Ad Hoc Networks. *Policies for Distributed Systems ond Networks, 2007. POLICY'07. Eighth IEEE Internotionol Workshop on*, 256–260.

Clarke Jr., E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking.* The MIT Press.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., & Robby. (2002). Expressing checkable properties of dynamic systems: The bandera specification language. *Internotionol Journol on Softwore Tools for Technology Tronsfer (STTT)*, *4*(1), 34–56.

Cossentino M., Gaglio S., Henderson-Sellers B., and Seidita V.: A metamodelling-based approach for method fragment comparison. In: *Proceedings of the 11th Internotionol Workshop on Exploring Modeling Methods in Systems Anolysis ond Design* (EMMSAD 06). Luxembourg, June 2006.

Damianou, N., Dulay, N., Lupu, E. C., & Sloman, M. (2000). Ponder: a language for specifying security and management policies for distributed systems. *Imperiol College Reseorch Report DoC 2000/1.*

DeLoach, S.A. (2009). Organizational Model for Adaptive Complex Systems. in Virginia Dignum (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global: Hershey, PA.

DeLoach S.A., and Valenzuela Jorge. L.: An Agent-Environment Interaction Model. in L. Padgham and F. Zambonelli (Eds.): *AOSE 2006*, LNCS 4405, pp. 1-18, 2007. Springer-Verlag, Berlin Heidelberg 2007.

DeLoach S.A., Oyenan W.H., and Matson, E.T. A Capabilities Based Model for Artificial Organizations. *Journol of Autonomous Agents ond Multiogent Systems*. Volume 16, no. 1, February 2008, pp. 13-56. DOI: 10.1007/s10458-007-9019-4..

DeLoach, S. A. (2002). Modeling organizational rules in the multi-agent systems engineering methodology. In Advances in artificial intelligence: 15th conference of the Canadian society for computational studies of intelligence (AI 2002) (Vol. 2338, pp. 1–15). Springer-Berlin/Heidelberg.

DeLoach, S. A. (2007). Developing a multiagent conference management system using the O-MaSE process framework. In M. Luck (Ed.), *Agent-oriented softwore engineering viii: The 8th internotionol workshop on ogent oriented softwore engineering* (Vol. LNCS 4951, p. 171-185). Springer-Verlag: Berlin.

DeLoach, S. A., Oyenan, W., & Matson, E. T. (2007). A capabilities based theory of artificial organizations. *Journol of Autonomous Agents ond Multiogent Systems.*

DiLeo, J., Jacobs, T., & DeLoach, S. (2002). Integrating ontologies into multiagent systems engineering. In *Fourth internotionol conference on ogent-oriented informotion systems (oios-2002).* CEUR-WS.org.

Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 internotionol conference on softwore engineering.* IEEE.

Ferber, J., Gutknecht, O., Jonker, C.M., Müller, J.P., and Treur, J. (2002). Organization Models and Behavioral Requirements Specification for Multi-Agent Systems, *Proc. of the 10th Europeon Workshop on Modeling Autonomous Agents in o Multi-Agent World*. Lecture Notes in AI, Springer Verlag.

Firesmith, D.G., and Henderson-Sellers, B.: *The OPEN Process Fromework: An Introduction.* Addison-Wesley, Harlow–England (2002).

Fleming, Q. W. and Koppelman, J. M. 2006 Earned Value Project Management. Project Management Institute.

Garcia-Ojeda, J. C., DeLoach, S. A., Robby, Oyenan, W. H., & Valenzuela, J. (2007, May). *O-MaSE: A customizable approach to developing multiagent development processes.* Honolulu, HI.

Garcia-Ojeda, J.C., DeLoach, S.A. Robby, Oyenan, W.H., & Valenzuela, J. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. In Michael Luck (eds.), Agent-Oriented Software Engineering VIII: The 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007), LNCS 4951, 1-15, Springer-Verlag: Berlin. 2008.

Harmon, S. J., DeLoach, S. A., & Robby. (2007). Trace-based specification of law and guidance policies for multiagent systems. In *8th annual international workshop on engineering societies in the agents world (esaw).*

Harmon, S. J., DeLoach, S. A., Robby, & Caragea, D. (2008). Leveraging organizational guidance policies with learning to self-tune multiagent systems. In *The second ieee international conference on self-adaptive and self-organizing systems (saso).*

Harmon, S.J., DeLoach S.A., and Robby. Trace-based Specification of Law and Guidance Policies for Multiagent Systems. *The Eighth Annual International Workshop "Engineering Societies in the Agents World"* (ESAW 07) Athens, Greece, October, 2007.

Henderson-Sellers, B., and Giorgini P. (eds.): *Agent-Oriented Methodologies*, Idea Group Inc., 2005.

Henderson-Sellers, B.: Process Metamodelling and Process Construction: Examples Using the OPEN Process Framework (OPF). *Annals of Software Engineering*. 14, 1-4 (2002) 341–362.

ISO. (1991). Iso/iec: 9126 information technology-software product evaluation-quality characteristics and guidelines for their use. International Organization for Standardisation (ISO).

Jong, N. K., & Stone, P. (2007). Model-based function approximation for reinforcement learning. In *The sixth international joint conference on autonomous agents and multiagent systems.*

Kagal, L., Finin, T., & Joshi, A. (2003). A policy based approach to security for the semantic web. In *The semanticweb - iswc 2003* (Vol. 2870, pp. 402–418). Springer-Berlin/Heidelberg.

Kelly, S., Lyytinen, K., and Rossi, M. MetaEdit+ : A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Advanced Information System Engineering. Springer Berlin / Heidelberg, 1996, 1–21.

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer, 36*(1), 41–50.

Kleppe, A., Warmer, J., and Bast, W. MDA Explained: The Model Driven Architecture – Practice and Promise. Adisson-Wesley. 2003.

Kok, J., & Vlassis, N. (2006). Collaborative Multiagent Reinforcement Learning by Payoff Propagation. *The Journal of Machine Learning Research*, *7*, 1789–1828.

Ligatti, J., Bauer, L., & Walker, D. (2004). Edit automata: Enforcement mechanisms for run-time security policies. In *International journal of information security* (Vol. 4, pp. 2–16). Springer-Verlag.

Luck, M., McBurney, P., Shehory, O., and Willmott, S.: Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink (2005).

MACR Lab. (2009). *Human Robot Teams.* `http://macr.cis.ksu.edu/hurt/`.

Manna, Z., & Pnueli, A. (1991). The temporal logic of reactive and concurrent systems: Specification. Springer-Verlag.

Mari, M., Poggi, A., Tomaiuolo, M., & Turci, P. (2006). A Content-Based Information Sharing Multi-Agent System. *Modelling Approaches*.

Matson, E., DeLoach, S.A. (2003) Capability in Organization Based Multi-agent Systems, Proceedings of the Intelligent and Computer Systems (IS '03) Conference, 2003.

Miller, M. (2007). *A goal model for dynamic systems.* Unpublished master's thesis, Kansas State University.

Miller, M.A. *Goal Model for Dynamic Systems*. Master's Thesis, Dept. of Computing and Information Sciences, Kansas State University, 2007.

Nair, R., Tambe, M., Yokoo, M., Pynadath, D., & Marsella, S. (2003). Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 705–711.

Odell J., Parunak V. D., and Bauer B.: Representing Agent Interactions Protocols in UML. In: Ciancarini, P., and Wooldridge, M. (eds.): *Agent Oriented Software Engineering*. LNCS 1957. Springer-Verlag, Berlin Heidelberg New York (2001) 121–140.

Olender, K., & Osterweil, L. (1990). Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, *16*(3), 268–280.

Panait, L., & Luke, S. (2005). Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, *11*(3), 387–434.

Paruchuri, P., Tambe, M., Ordóñez, F., & Kraus, S. (2006). Security in multiagent systems by policy randomization. In *Aamas '06: Proceedings of the fifth international joint conference on autonomous agents and multiagent systems* (pp. 273–280). New York, NY, USA: ACM Press.

Peña, J., Hinchey, M. G., & Sterritt, R. (2006). Towards modeling, specifying and deploying policies in autonomous and autonomic systems using an AOSE methodology. *EASE*, *0*, 37–46.

Peshkin, L., Kim, K., Meuleau, N., & Kaelbling, L. (2001). Learning to Cooperate via Policy Search. *Arxiv preprint cs.LG/0105032*.

Robby, DeLoach, S.A., and Kolesnikov, V.A. (2006). Using Design Metrics for Predicting System Flexibility. In: Baresi, L, and Heckel, R (eds.): *Fundamental Approaches to Software Engineering*. LNCS 3922. Springer-Verlag, Berlin Heidelberg New York (2006) 184–198.

Robby, Dwyer, M. B., & Hatcliff, J. (2003). Bogor: An extensible and highly-modular model checking framework.

Robby, Dwyer, M.B., & Hatcliff J.: Bogor: A Flexible Framework for Creating Software Model Checkers. In: *Proceedings of the Testing: Academic & industrial Conference on Practice and Research Techniques*. IEEE Comp Society, Washington, DC, 3–22.

Scott A. DeLoach, Mark F. Wood and Clint H. Sparkman, Multiagent Systems Engineering, The *International Journal of Software Engineering and Knowledge Engineering*, Volume 11 no. 3, June 2001.

Seidita, V., Cossentino, M., Gaglio, S.: A repository of fragments for agent systems design. In: *Proceedings of the 7th Workshop from Objects to Agents* (WOA06). Catania, Italy (2006) 130–137.

Shoham, Y., & Tennenholtz, M. (1995). On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, *73*(1-2), 231-252.

Smith, R. L., Avrunin, G. S., Clarke, L. A., & Osterweil, L. J. (2002). Propel: an approach supporting property elucidation. In *Icse '02: Proceedings of the 24th internotionol conference on softwore engineering* (pp. 11–21). New York, NY, USA: ACM Press.

Stoller, S. D., Unnikrishnan, L., & Liu, Y. A. (2000). Efficient detection of global properties in distributed systems using partial-order methods. *Computer Aided Verification: 12th International Conference Proceedings*, *1855/2000*, 264–279.

Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., & Lott, J. (2003). Kaos policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Policy 2003: Ieee 4th internotionol workshop on policies for distributed systems and networks* (pp. 93–96). IEEE.

van Lamsweerde, A., Darimont, R., and Letier, E. Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering, special Issue on Managing Inconsistency in Software Development 24, 11 (Nov. 1998), 908 – 92.

Viganò, F., & Colombetti, M. (2007). Symbolic Model Checking of Institutions. *Proceedings of the 9th Internotionol Conference on Electronic Commerce.*

Viganò, F., & Colombetti, M. (2008). Model checking norms and sanctions in institutions. *Coordinotion, Orgonizotions, Institutions, and Norms in Agent Systems III*, 316–329.

Watkins, C. J. (1989). *Leorning from Deloyed Rewords.* Unpublished doctoral dissertation, Cambridge University.

Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2001). Organisational rules as an abstraction for the analysis and design of multi-agent systems. *Internotionol Journol of Softwore Engineering ond Knowledge Engineering*, *11*(3), 303–328.

Zhong, C., & DeLoach, S. A. (2006). An investigation of reorganization algorithms. In *Proceedings of the internotionol conference on ortificiol intelligence (ic-oi'2006)* (p. 514-517). CSREA Press.

Appendix A Appendix A. O-MaSE Definition

7 WORK UNITS - TASKS

7.1 Model Goals

Purpose	Transform initial system requirements into a set of structured system goals.

Steps

Identify Goals. The aim of this step is to capture the essence of an initial set of requirements. That is, the Goal Modeler extracts scenarios from the initial specification; and then, the Goal Modeler describes the goal of each particular scenario.

Decompose Goals. The Goal Modeler must structure/decompose the goals into a Goal Model. The Goal Model is organized as a tree, with the top-level goal encompassing the overall goal of the system, while lower-level goals decompose the higher-level goals. In order to complete this step, the Goal Modeler must accomplish the next two sub-steps:

Identify the overall system goal. It is often that a single system goal cannot be directly extracted from the group of goals captured at step *Identify Goals*. In that case, the Goal Modeler must summarize the highest-level goals into an overall system goal. Otherwise, the overall goal is placed at the top of the Goal Model.

Derive Goals. Once the overall system goal is in place, goals may be decompose into new sub-goals. Each sub-goal must support its parent goal in the hierarchy goal and defines ***what*** must be done to accomplish the parent goal. If as result of the goal decomposition the sub-goals prescribe something that has to be done, the Goal Modeler has found the operations/functions that would be carried out by agents in run-time. Otherwise, the Goal Modeler will continue this step until any further decomposition would result in operations/functions instead of goals (i.e., the Goal Modeler prescribes ***how*** a goal should be accomplished).

Logically Structure Goals. In this step, the Goal Modeler determines whether – in order to achieve a parent goal – it is necessary to achieve all its sub-goals or not. The first case is known as an AND-decomposition (conjunctive), and the latter is known as an OR-decomposition (disjunctive). That is, some parent goals in the Goal Model (see page 94 for more detail) may require that its children must be achieved (AND), while other goals may have alternative ways to be achieved (OR).

Guidelines

The *initial system context*, starting point of O-MaSE, is usually the system description or the software requirement specification with a well-defined set of requirements. These requirements tell the Goal Modeler the goals that the system must achieve and how the system should or should not behave based on the inputs to system and its current state.

Because the goals encapsulate the critical system requirements, the Goal Modeler should specify abstractly – the goals – as possible without losing the spirit of the requirement. This abstraction can be achieved by removing detailed information when specifying the goals. Once the goals have been captured, they provide the foundation for further system analysis.

In order to map the goals in the Goal Model, the Goal Modeler must study the goals in terms of their importance and inter-relationships. Even though goals have been captured, they are of various importance, and level of detail. Then, the Goal Modeler must identify the overall system goal, which is placed at the top of the Goal Model. However, sometimes is the case that a single goal cannot be extracted directly from the requirements. In such case, the highest-level goals are summarized to create an overall system goal. Once the basic Goal Model is in place, goals may be decomposed into new sub-goals. Each sub-goal must support its parent goal in the hierarchy and defines what must be done in order to accomplish the parent goal. This process continues until the goals cannot be decomposed. Each level of decomposition can be either disjunctive or conjunctive. A disjunctive high-level goal is satisfied when a single sub-goal (child) is satisfied, whereas a conjunctive high-level goal is satisfied when all of its sub-goals (children) are satisfied.

Finally, the identification and goal decomposition is a non-trivial, but critical, task. Thus, the Goal Modeler and the Domain Expert must interact a lot with the client in order to come out with a robust Goal Model.

7.2 Goal Refinement

Purpose

This purpose of this task is to capture the dynamism associated with the system and the goals that it has to achieve. This is accomplished by defining all the relationships and attributes for each goal in the Goal Model.

Steps

Identify Goal Precedence. To capture the dynamic nature of the system, the Goal Modeler must identify the time-based relationship that exists between goals. This full or partial ordering of goal execution in the systems is called "Goal Precedence." The Goal Modeler must identify any time-based precedence relations between goals in the Goal Model. This "precedes" relation from goal x to goal y states that goal *y* cannot be pursued until goal *x* has been achieved.

Identify Goal Instantiation Triggers. As events in the system occur, the Goal Model must dynamically adapt to those events. In this step, the Goal Modeler must identify the important events that may occur during the pursuit of a goal that would cause the creation of new, related goals. For example, if an event *e1* can occur in the pursuing of goal *g1* that instantiates goal *g2*, we say that, *g1 triggers g2*.

Parameterize Goals. When a goal has been identified as a "triggered" goal, the Goal Modeler must generally parameterize the goal to allow the goal to take on a context sensitive meaning. Since the system generally cannot control triggering events, several instances of goal may be instantiated based on the specific situation surrounding the triggering event. For instance, assume that goal *g1* requires an agent to wait for some instructions. When the instructions arrive, the agent forwards this event (the receipt of instruction) to the organization, which causes the instantiation of a new goal of type *g2*. This new g2 instance carries out the instructions and thus must be parameterized based on the instruction received. If a second instruction arrives, a second instance of goal *g2* is instantiated with its specific instruction as its parameter.

Guidelines

While it is typically more efficient to wait until a complete Goal Model is available, the nature of software development will generally necessitate going back and forth between the Model Goals task and the Goal Refinement task. It is very important that Goal Modeler performing the Goal Refinement task be thoroughly familiar with the initial system context in order to understand the dynamic nature of the system.

Because this task introduces the concepts of *precedes, triggers,* and *parameterize*. It is critical that the Goal Modeler understands the differences and relations between them. That is, goal precedence specifies that one goal must be achieved before another goal can be started while goal triggering is based on events that occur during system operation. A single instance of un-triggered goals is created at system initialization, while other triggered goals are created when specific events occur. Triggered goals should almost always be parameterized to define the context of the goal. The only time triggered goals are not parameterized is when the same goal should be re-accomplished.

7.3 Model Organization Interfaces

Purpose

The objective of this task is to identify organization's interfaces with external actors.

Steps

Identify External Actors. The Organization Modeler must identify each external actor that may interact with the Organization (system). Such actors can be humans, other software systems, and hardware devices (e.g., robots). This information can be extracted from the initial system context documentation.

Identify Interactions with External Actors. The Organization Modeler must identify and name all the possible ways that organization may interact with external actors. That is, the Organization Modeler must capture the flow of information coming into or out of the organization. This flow of information is captured by means of protocols between the organization and the external actors. The Organization Modeler may document the detailed description of each interaction (if available) by means of Protocol Models. If a detailed description is not currently available it may be added later as these protocols will be mapped to protocols later in the design.

Identify the goal to be achieved by the organization. The Organization Modeler must identify the goal to achieve by the organization. In general, the goal to achieve is provided by the top-level goal depicted in the Goal Model.

Guidelines

Based on the information described on the initial system context, the Organization Modeler must extract all the details regarding the organization, the external actors, and their interactions. Thus, for every possible interaction, the Organization Modeler must depict them in terms of protocols. Protocols are represented by means of arrows. Arrow represents the flow of information from the initiator of the interaction to the responder. The initiator and responder of a protocol must be either an external actor or the organization.

7.4 Model Roles

Purpose

This task focuses on identifying all the roles in the organization, as well as, their interactions with each other and with external actors.

Steps

Identify the roles required by the organization. In this step, the Role Modeler must identify the roles necessary to achieve the goals defined in the Refined Goal Model. An easy way to perform this step is to identify the basic skills required by the organization to achieve its leaf goals. Generally, roles are identified to achieve specific leaf goals of the refined Goal Model. In general, the Role Modeler must define at least one role that can achieve each goal. However there are reasons where it is useful to have a single role for multiple goals or multiple roles to achieve a single goal. These reasons include convenience and/or efficiency.

Identify the role's interactions. In order to carry out the interactions defined between the organization and the external actors, the Role Modeler must map these interactions down to specific roles. Also, it is often the case that several roles must interact in order to achieve their individual goals. In the Role Model, these interactions are simply identified as information flows, which are captured as protocols between roles or between roles and external actors. The Role Modeler may document the detailed description of each interaction (if available) by means of Protocol Models.

Identify the role's capabilities. In O-MaSE a capability represents a way which agents can sense and effect the environment. Therefore, a role must require a capability in order to achieve an assigned goal. Thus, the Role Modeler must identify a set of capabilities required by roles to achieve the assigned goals. These capabilities may include hard capabilities such as sensors and effectors as well as soft capabilities such as software functions or plans.

Guidelines

The Model Roles task focuses on identifying the roles in the organization and their interactions with each other.

All external actors from the Organization Model should show up as external actors in the organization's Role Model and the protocols between external actors and the organization must be mapped to protocols between external actors and the specific roles in the system. Thus, the Role Model is a refinement of the Organization Model.

In addition, each leaf in the Goal Model must be assigned to a role in the Role that can achieve it, as denoted by the «achieves» designator in the body of the role. Thus, each role should achieve at least one leaf goal, although in general, a role may achieve multiple leaf goals.

Because roles require capabilities in order to achieve goals, the Role Modeler must identify such capabilities. Basically, the required set of capabilities depends on the requirements of the goal. For instance, if role *x* achieves goal *transport object*; it is clear that role *x* would typically requires the capability of movement.

7.5 Define Roles

Purpose

This task further refines the capabilities required for an agent to play a role, and the constraints and permissions that apply to the agents playing that role.

Steps

Role Description. In this step the Role modeler, must extend the roles' description in terms of the capabilities required by the role and the permissions and constraint associated with the role. Also, if needed, the Role Modeler may further refine a role by means of an internal set of sub-goals.

Guidelines

The capabilities are identified based on the goals that the role seeks achieve. Generally, a role requires at least on capability in order to achieve a goal. One approach to defining a role is to prescribe a specific plan to achieve that role.

Since plans are a type of capability that the role may require, this is typically carried out by defining a plan via a Plan Model and then creating a capability that performs such plan (see Model Capabilities and Model Plans tasks).

Role refinement provides a less restrictive set of guidelines (in terms of sub-goals) on what has to be done in order to play the role. However, sub-goals do not prescribe any specific process. This model may be defined using the Model Goals and Refine Goals tasks as defined for organizations.

Depending on the goal achieved, access permissions to certain resources may be granted to agents playing that role.

7.6 Model Domain

Purpose

This task captures the objects, relationships, and behaviors that define the domain in which agents would sense and act.

Steps

Identify objects' types. The Domain Modeler must understand exactly what types of objects may be in the domain (environment). Such objects are defined simply via a name and a set of attributes. In O-MaSE, objects are closely related to object-oriented classes or types and do not represent object instances. All agents identified in the organization are also objects within the Domain Model.

Identify the relationships between objects. Because the Capability Model needs to define the actions that an agent may perform upon objects in the domain, the Domain Modeler must identify all the possible relationships between the different objects in the domain and the agents. These relations determine which agent actions may affect which objects.

Identify the principles and processes that govern the domain. Besides identifying objects and attributes, it is also important to identify the principles and processes that govern the environment. The principles and processes are defined in terms of autonomous processes that cause actions upon domain model objects.

Guidelines

The domain model is developed using traditional domain modeling or domain analysis techniques common to most object oriented development methodologies.

In order to define the actions that an agent may perform upon then environment, it is critical that we understand exactly what types of objects may be in the environment and the attributes of those objects.

In O-MaSE we use a simple Domain Model to model the objects upon which agents perform the basic actions of sensing and affecting the environment through interactions. Basically, the domain is a container of Objects, which can include Agents situated in the environment. All the objects in the domain are affected by physical Principles that are implemented by Processes. Objects are defined simply via a name and a set of attributes. Here, domain objects are actually more closely related to object-oriented classes or types than true object instances.

The Domain Modeler should identify all possible objects that would be found in the environment (included agents). For each object, the Domain Modeler must identify and document its attributes. Because objects share relationships within the environment, the Domain Modeler must identify and document such relationships. In some cases, such relationships can be exploited by means of capabilities (i.e., an agent may be able to manipulate some environment object, so there should be a relation between the agent and the object that can be used by some action possessed by the agent).

In order to identify the principles and processes that govern the environment, Domain Modeler must identify any physical property that can be mapped into the Domain Model. For example, in detecting radiation, there is the well-known principle that the amount of radiation intercepted varies as the square of the distance between the source and the sensor (a sensor can be seen as an agent in this case). This relation can be defined by the following equation:

$$radiation(x,y) = \sum_{\forall o:box} \frac{o.rad}{\sqrt{(o.x-x)^2+(o.y-y)^2}}$$

7.7 Model Capabilities

Purpose

The Model Capabilities task defines the internal structure of capabilities that an agent possesses in order to sense and to act in the domain.

Steps

Because capabilities can be implemented as either actions or plans, there two ways to model capabilities:

Actions:

Define action signatures. The first step in defining actions is to identify the actions associated with each capability. That is, the Capability Modeler must define action signatures based on the proposed use of the capability in the system. Often, these actions are identified during the Model Plans tasks where the overall approach to achieve a goal are defined. In the plan, actions are identified and then assigned to specific capabilities.

Define action effect. An action is defined as a single accessible operation via a set of pre and post-conditions over objects in the domain model. The preconditions determine whether or not the operation can be executed. If the preconditions do hold, the post-conditions determine the desired state of the world (objects in the domain) after the completion of the operation. Actions may also be defined as simple functions that do not directly affect objects in the domain; they are used to simply compute values based on set input parameters.

Plans:

Define Plan.   Plans are defined using the Model Plans task.

Guidelines

Capabilities can be used to define a plan or a set of actions that can be used by a plan to achieve some goal.

Action-based capabilities are defined in terms of actions performed on the environment or as functions on a set of input values. The Capability Modeler can also use aggregation to compose capabilities using the actions defined by other capabilities. Also, the Capability Modeler can capture the capability of an agent to send and receive messages.

The Model Capabilities task also allows the designer to create new capabilities out of existing capabilities. For example, a robotic Rescue capability may be defined as a composed capability that uses the previous defined capabilities of Search, Pickup, and Communicate. Using aggregation, the Rescue capability has access to all the actions defined as part of the Search, Pickup, and Communicate capabilities. The actions from the sub-capabilities may be used directly by the agent possessing the Rescue capabilities. In addition, the Capability Modeler may define more complex actions in the Rescue capabilities using the actions of the sub-capabilities.

7.8 Model Agent Classes

Purpose

The purpose of the Model Agent Classes task is to identify the type of agents that will participate in the organization.

Steps

Mapping roles to agent classes. If the O-MaSE process has defined a Role Model, the Agent Class Modeler defines specific types of agents based on the Roles that need to be played in the Role Model. The modeler may define a *plays* relation between agent classes and roles or may allow the capabilities of the individual agents to determine what roles it can play at runtime.

Identify Capabilities. In this step, the modeler identified the capabilities possessed by the agent classes. These capabilities do not have to be fully defined. If a Role Model exists, the modeler needs to ensure that the agent classes defined to play specific roles possess all the capabilities required for those roles. In a system without a role model or in a system with predefined agent classes, the modeler may simply define the capabilities possessed by each agent class.

Map protocols between agents. During this step, the Agent Class Modeler must identify the protocols in which agent classes must participate. These protocols are derived from the interaction of the agent's assigned roles, if assigned.

Guidelines

While the Agent Class Modeler makes the assignment of roles to agents, typical software engineering concepts such as coupling and cohesion should be used to evaluate the assignment. That is, agent can play a given role.

Also, any protocol documented in the Role Model must be documented in this task. Also, each agent has to be attached to at least one. Because every agent is described in an abstract way (i.e., roles played, capabilities possessed, and protocol it participates in), the details of the design must be completed by means of tasks such as Model Protocols and Model Plan tasks.

7.9 Model Protocols

Purpose

This task defines the interactions between either agents playing roles or roles.

Steps

Identify Interactions. The Protocol Modeler identifies interactions between either agents playing roles or roles. The protocols should already be identified in associated organization, role, or agent class models. Also, the Protocol Modeler must identify the constraints on the passing message between those entities (i.e., agents or roles).

Define Interaction Details. In this step, the Protocol Modeler must define and document the details of the messages that make up the protocol. The details of the protocol include message names and parameter types, the sequencing of messaging and any possible alternative sequences.

Guidelines

For each protocol identified in organization/role/agent class models, the Protocol Modeler must define individual Protocol Models. These protocols are typically modeled using models similar as UML/AUML Interaction Diagrams, which allow the Protocol Modeler to specify message sequence, alternatives, loops, and references to other protocols.

Often, the system designers may wish to use pre-defined protocols such as the *Request*, *Query*, or *Contract Net* FIPA multiagent protocols defined at http://www.fipa.org/repository/.

If a set of plans have already been implemented that require the interaction of two agents/roles, the protocols defined must adhere to the constraints of the pre-existing plans. Obviously, an iterative process between defining protocols and plans may be used.

7.10 Model Plans

Purpose

This task represents a mean by which agents can satisfy a goal in the organization. A plan can be seen as an algorithm for achieving a specific goal. Also, plans are made of several actions or sub-plans and may require interaction with other agents though some defined protocols.

Steps

Define actions. In this step, the Plan Modeler determines the actions necessary to achieve the goals of the system. These actions are then assigned to states within the plan.

Capture message iterations. The Plan Modeler captures the messages either sent or received by an agent in carrying out the overall plan. These messages should be consistent with protocols defined between actions, organizations, roles, and agents in associated organization/role/agent class models.

Guidelines

Agent plans are created to achieve specific goals, or sets of goals. These plans may be associated with roles or agents via capabilities.

Depending on the internal architecture chosen for each agent, we could develop multiple Agent Plan Models for each agent. This might be the case when we wanted a unique plan for each role an agent could play or if we could choose between multiple plans to achieve the same goal. In either case, the Agent Plan Modeler would be responsible for selecting the appropriate plans and interleaving their execution if required.

Plans are typically modeled using finite state automata to specify a single thread of control that defines the behavior that the agent should exhibit. Plans specify both actions an agent can perform and messages an agent may send or receive. The interplay between the actions and interactions with external agents defines the overall plan.

7.11 Model Policies

Purpose — The Model Policies task defines a set of formally specified rules that describe how an organization may or may not behave in particular situations.

Steps — **Identify system events of interest.** The Policy Modeler must identify such events based on the following criteria:

Event	Definition
$C(g_i)$	Goal g_i has been completed (achieved)
$T(g_i)$	Goal g_i has been triggered.
$A(a_i,r_j,g_k)$	Agent a_i has been assigned role r_j to achieve goal g_k.

Identify properties of interest. The Policy Modeler must identify some properties that may be domain specific (only relevant to the current system), and others may be system properties such as the number of roles an agent is currently playing.

Specify assignment policies. The Policy Modeler must specify policies concerning agent assignments to roles in order to constrain the set of possible assignments. This can reduce the search space when looking for the optimal assignment set.

Guidelines — Policies can be incrementally defined. That is, Policy Modeler can start defining policies by taking the Goal Model. Yet, it is recommended to wait until goals, roles, and agent has been defined. This is an important fact, because the Policy Modeler can use the information regarding the relation between goals, roles, and agents to document the rules that would govern the organization in running time.

7.12 Model Actions

Purpose — This task captures the interactions performed by agents with objects within the environment.

Steps — **Identify actions.** The modeler must first identify the actions that need to be performed by agents in the system. Actions are typically extracted from Protocol Models, Plan Models, or Capability-Action Models. The modeler may also examine the Domain model to determine likely actions that will need to be performed on domain entities.

Define action signatures. Here, the modeler must define the name, and the input and output types of the action. Again, inputs may be extracted directly from the Protocol or Plan Models while outputs can be extracted directly from Plan Models. If these models are not available, the modeler must determine the likely inputs outputs required by the system agents. Note that as the modeler models the pre- and post-conditions in the next step, it is likely that new inputs and outputs may be identified.

Define pre- and post-conditions. The modeler defines the action's pre- and post-conditions in terms of the actions effect on Domain Model entities. Actions may read and write accessible entity attributes defined in the Domain Model.

Guidelines — Actions may only be fully defined once at least one capability has been identified and the domain model created. The reason is that actions must be attached to a capability and they are defined over the domain model entities.

For simplicity of identifying actions, it is recommended that either the protocol or plan model be defined prior to actually identifying and defining the actions. This allows the modeler to see how the actions will be used.

If a modeler is defining an action for a predefined piece of software or hardware, the action definition must be consistent with the software/hardware being modeled. In this case, the action definition may take place before a protocol or plan model is defined.

For automated checking and verification, pre- and post-conditions should be specified formally. If no automatic verification tools are being used, the modeler may use pseudo-code or human language pre- and post-conditions.

8 Work Units - Techniques

8.1 AND/OR Decomposition

Typical tasks for which this is needed: Model Goals.

Technique description: Some goals in the Goal Model may require that its children must be achieved (AND); other goals may have alternative ways to be achieved (OR). When constructing the Goal Model, the AND parent-children relation is represented using diamond notation and to represent and OR relation a triangle notation is used. Once the goals have been identified from the system requirements or from a Role Description Document, and the Goal Model has been created by decomposing each goal in sub-goals as required, representing them using rectangles, the goal to sub-goal relation can be modified using diamond notation to show the goals that require a composition of its children to be achieved, and leaving the rest as is, to represent the OR refined parent goals, which may have alternative ways to be achieved. For example, in Figure 50 we represent a Goal Model. The syntax uses standard rectangle notation with the keyword <<goal>>. The aggregation notation is used to denote AND refined goals (conjunction), whereas the generalization notation is used to denote OR refined goals (disjunction).

Deliverables: Goal Model

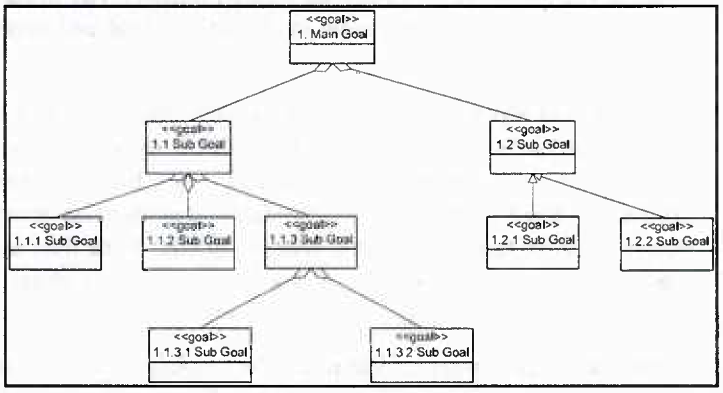


Figure 50. Goal Model

8.2 Attribute-Precede-Trigger Analysis (ATP Analysis)

Typical tasks for which this is needed: Goal Refinement

Technique description: To complete the refined Goal Model, the goal to sub-goal relation, system requirements, and role definition are reevaluated to identify any possible way to optimize the Goal Model by including parameters in any appropriated goal. Also, the Goal Model is analyzed to identify subsets of goals that can be achieved at different times to allow the organization to work on one part of the tree at the time. For instance, Figure 51 depicts a refined Goal Model. Basically, a refined Goal Model introduces the relations of precedes and triggers. *Precedes* relation determines which goals must be achieved before a given goal may be attempted. This relation is mapped with the type designator <<precedes>>. On the other hand, the *triggers* relation is similar to precedes in that it restricts specific goals from being pursued until a specific event occurs. For instance, assume that goal 1.1.1 triggers goal 1.1.2 based on event *e0()*. Instead of requiring the system to achieve goal 1.1.1 before pursuing goal 1.1.2, the trigger relation requires the system to instantiate a new instance of goal 1.1.2 when event *eo()* occurs during the achievement of goal 1.1.1.

Deliverables: Goal Model.

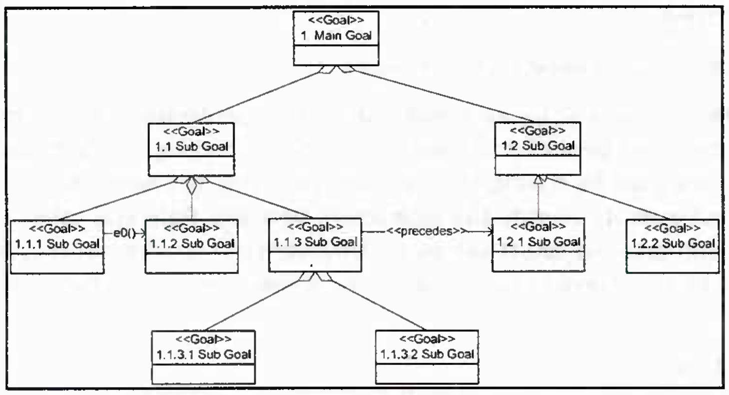


Figure 51. Refined Goal Model

8.3 Organizational Modeling

Typical tasks for which this is needed: Organization Modeling.

Technique description: This technique helps analyze the system in a top-bottom way, and both, inside-out and outside-inside. Create a diagram for the top goal in the Goal Model or for the top goal for a Role identified in the Role Definition task. Identify each external actor that may interact with the Organization, integrate these actors to the diagram. Establish the relation between the Organization and the Actors setting and naming the Interaction protocols between them (see Figure 52).

Deliverables: Organization Model

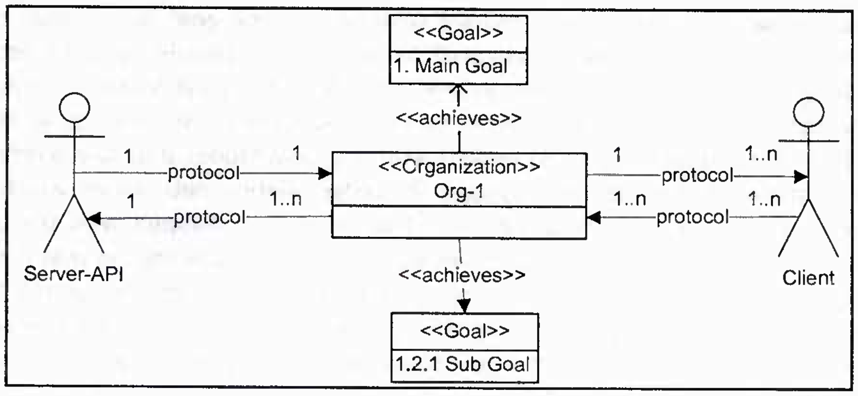


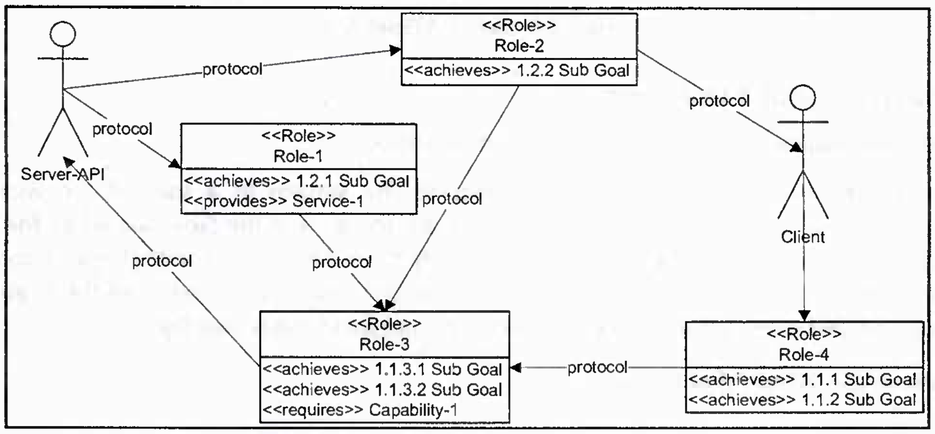
Figure 52. Organization Model

8.4 Role Modeling

Typical tasks for which this is needed: Model Roles.

Technique description: Generally, for each leaf goal in the Goal Model, create a role that can achieve it. However, it might sometimes be useful to have a single role achieving multiple goals. Also, more flexible organization can be designed by having several roles achieving the same goal. Once each goal has at least one role that achieves it, identify the interaction between Role and other Roles and/or external Actors. Interactions with external actors can be derived from the requirements or from the Organization Model if provided. Add a protocol between two roles if they need to exchange some information (see Figure 53).

Deliverables: Role Model.



<<Role>>
Role-2
<<achieves>> 1.2.2 Sub Goal
<<Role>>
Role-1
<<achieves>> 1.2.1 Sub Goal
<<provides>> Service-1
<<Role>>
Role-3
<<achieves>> 1.1.3.1 Sub Goal
<<achieves>> 1.1.3.2 Sub Goal
<<requires>> Capability-1
<<Role>>
Role-4
<<achieves>> 1.1.1 Sub Goal
<<achieves>> 1.1.2 Sub Goal
Server-API
Client
protocol
protocol
protocol
protocol
protocol
protocol
protocol

Figure 53. Role Model

8.5 Role Description

Typical tasks for which this is needed: Role Definition.

Technique description: *Technique description:* To perform a Role Description, the following actions can be performed for each role:

Identify all the capabilities required by that role. Capabilities are identified based on the goals that the role can achieves. All roles should require at least one capability.

If needed, the role can be further decomposed into sub-goals. Intuitively, sub-goals provide guidelines on what has to be done in order to play the role. However, sub-goals do not prescribe any specific process.

Identify the permissions. Depending on the goal achieved, access permissions to certain resources need to be granted only to agents playing that role.

Deliverables: Role Description Document.

Identify the constraints. Role	Name	Achieves	Requires	Constraints
Role-1	Role-1	1.2.1 Sub Goal	n.a.	Add constraints here
...	...	...	...	...

Figure 54. Role Description Document (excerpt)

8.6 Agent Classes Modeling

Typical tasks for which this is needed: Model Agent Classes

Technique description: Once the roles have been identified, create at least one agent class having all the capabilities required to play each role. An Agent Classes is a template for a type of agent in the system and is analogous to object classes. An Agent class identifies the capabilities that it possesses. Each capability is designated by the type designator <<capability>>. Agent classes can also be defined in term of the role they can play, denoted by the type designator <<plays>>. Likewise, the <<possesses>> relation between agent classes and capabilities (denoted by the <<capability>> type designator) indicates the capabilities possessed by instances of that class of agent. Furthermore, the arrows represent interaction between agents and actors, and between agents (see Figure 55).

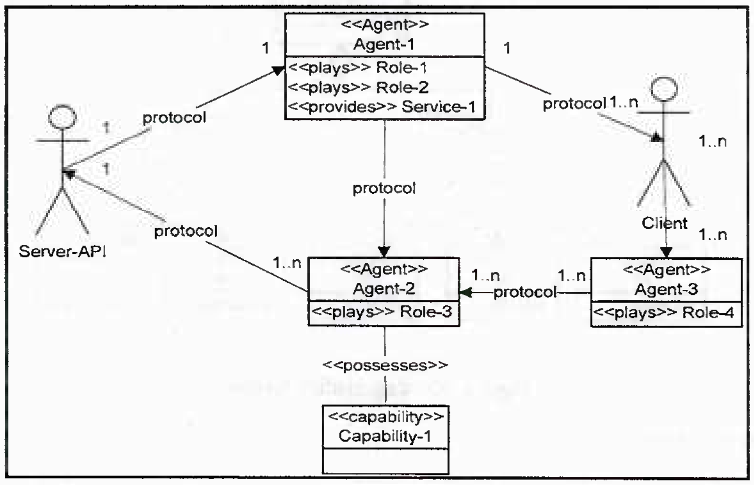


Figure 55. Agent Classes Model

Deliverables: Agent Classes Model.

8.7 Protocol Modeling

Typical tasks for which this is needed: Model Protocol

Technique description: Once the interactions between agents or roles have been identified, the next step is to follow the current AUML protocol diagram specification (Odell, Parunak & Bauer 2001). Using this technique we map the agent(s)/role(s) participating in a protocol, as well, the messages passed between them (see Figure 56).

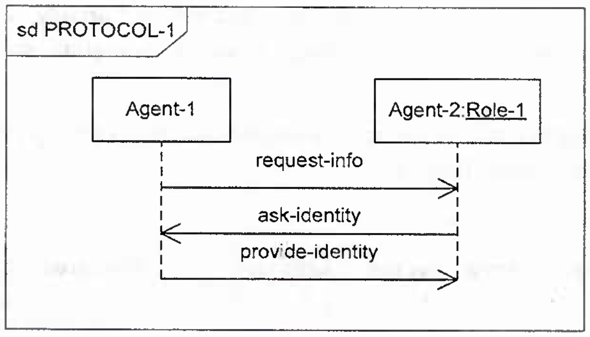


Figure 56. Protocol Model (simple example)

Deliverables: Protocol Model.

8.8 Capability Modeling

Typical tasks for which this is needed: Model Capabilities

Technique description: This technique is based on traditional class diagrams with the particularity that a Capability Model may include three types of classes: Action, Protocol, and Plan. An action capability is an atomic ability an Agent may possess. A protocol capability is a set of steps and conditions under certain actions are executed. A plan capability is a set of capability actions and protocols that is defined in order to achieve one or more goals. Basically, a *Capability Model* defines capabilities in terms of other capabilities as well as atomic actions (see Figure 57).

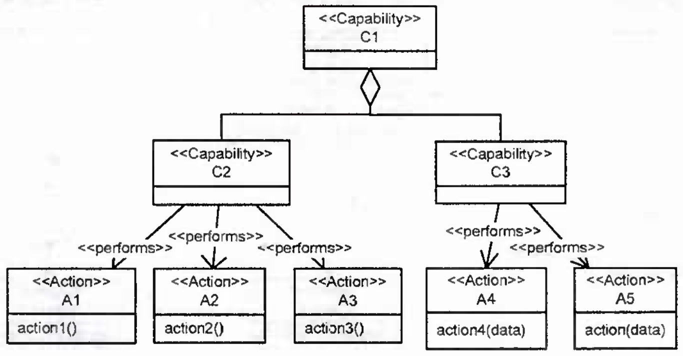


Figure 57. Capability Model

Deliverables: Capability Model.

8.9 Plan Specification

Typical tasks for which this is needed: Model Plans.

Technique description: This technique captures actions and protocols used by an agent to achieve a goal using a Finite State Automata (FSM). This technique is based on traditional state diagrams and consists of state and transitions representing the internal state and communications protocols of an agent (see Figure 58).

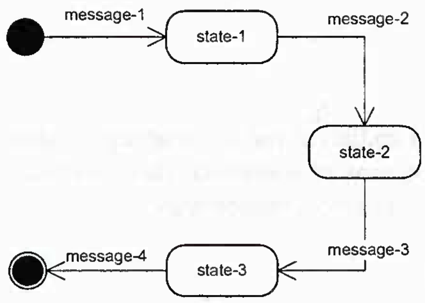


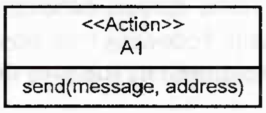
Figure 58. Agent Plan Model

Deliverables: Agent Plan Model.

8.10 Action Analysis

Typical tasks for which this is needed: Model Actions

Technique description: The Action Model is used to depict the different actions that an Agent can perform, and also a text document is included to define any pre and post-condition an action may have. Each action is labeled with the type designator <<Action >> and the attributes are also defined in the notation. Furthermore a file with the pre and post condition for each action is documented. As result the *Action Model* work product is obtained (see Figure 59).



A1:	send(message, address)
Pre:	not(address = null)
Post:	self.possesses.communicates->select(add = address).q->includes(message)

Figure 59. Action Model

Deliverables: Action Diagrams.

8.11 Model Policies

Typical tasks for which this is needed: Model Policy

Technique description: During the organization design, the Policy Modeler captures either all desired or required properties of the system and writes them in natural language. Once all the policies have been identified, formally specify them using a formal language. For example, the policy:

$$P1: \ \forall\ a1, a2: agent, r: role, a1.plays\ (r1)\ \wedge a1.plays\ (r2)\ \Rightarrow r1=r2,$$

depicts that an agent only can play a role at a time.

Deliverables: Policy Model.

9 WORK PRODUCTS

9.1 Goal Model

To capture the purpose or overall function of the artificial organization we use goals, and those goals are structured in the Goal Model by means of traditional class notation. Diamond and triangle notation is used to denote AND/OR goal decomposition, respectively.

9.2 Refined Goal Model

This is an extended version of the Goal Model. After executing the Goal Refinement task, the Refined Goal Model will include any attribute a Goal may have, and all goals precedence and triggering relationships. These relationships are represented with a directed arrow notation with the <<precedes>>, <<triggers>> type designators respectively.

9.3 Organization Model

The Organization Model is a diagram that captures the interaction between the organization and external actors. Traditional class notation is used to represent the organization and external actors. The relation between the organization and external actors denote interaction protocols.

9.4 Role Model

A role model captures all the roles in the organization along with the goal(s) they achieve and possible interaction protocols. Boxes represent roles. Arrows between actors and roles or between two roles denote the initiator/responder relationship. Following traditional class notation, goals achieved by those roles will be annotated as attributes and denoted by the type designator <<achieves>>.

9.5 Role Description

A Role Description Document is a textual description of each role. It captures the permissions and constraints associated with the role along with the sub-goals and capabilities required by that role.

9.6 Agent Classes Model

An Agent Classes Model defines the agent classes and sub-organizations that will populate the organization. The agent class is similar to an object class. Its attributes are the capabilities it possesses, and the roles it plays.

9.7 Agent Plan Model

An agent plan model follows traditional state diagrams (i.e., states, events, and transitions).

9.8 Capability Model

This model can include three levels of capabilities: Capability Action, Capability Protocol, and Capability Plan. A Capability Action is represented by means of traditional class notation. In addition, actions can be nested, that is, they can be composed by other Capability Actions. A Capability Protocol is a drawn similar to a traditional class identified with the protocol name. It can be nested, that is, it can be composed by other capability Protocol. A Capability Plan is a UML class identified by the plan name. This capability is composed by capability Actions, capability Protocols, and may or may not be include other capability Plans.

9.9 Protocol Model

A Protocol Model consists of Protocol Diagrams as specified in AUML. This model allows capturing alternative and repetitive message structures. Also, this model captures the different relations/interaction between two entities in an O-MaSE model such as in Role Model, Agent Model, or Organization Model.

9.10 Domain Model

A domain model contains the domain entities relevant to the organization, their attributes, and their relationships. This model uses traditional class notation to show the relationships between entities, producing a model similar to traditional class diagram.

9.11 Policy Model

A policy model is a document containing all the rules/constraints applicable to the system. These constraints are expressed by using First Order Predicate Logic.

9.12 Action Model

An Action Model is documented via pre- and post conditions for each Action. Each action defines at least one operation that implements the action effect/sense or ability an Agent has.

10 Producers

10.1 Goal Modeler

This role is responsible for the generation of the Goal Model and the Refined Goal Model. The Goal Modeler therefore has to be able to read and understand the system description/SRS and also be able to interact and maintain a clear an open communication with the Domain Experts/Customers. This person requires knowledge about techniques: AND/OR Decomposition, and ATP Analysis, to create, validate and maintain the Goal Model and the Refined Goal Model.

10.2 Organization Modeler

This role is responsible for the generation of the Organization Model. The Organization Modeler therefore has to be able to read and understand organization diagrams and also be able to interact and maintain a clear an open communication with the Goal Modeler/Customers. This person requires knowledge about the techniques: AND/OR Decomposition, and ATP Analysis.

10.3 Role Modeler

This role is responsible for creating the Role Model and the Role Description work products. The skills required for this role are: knowledge about the O-MaSE Role Model specification and general knowledge about the system under analysis-design. This role may have close interaction with the Goal Modeler and Agent Modeler.

10.4 Agent Classes Modeler

This role is responsible for creating the Agent Class Model. This role requires skills and knowledge about the O-MaSE Agent Class Model specification. In addition, this role may interact with the Role Modeler and the Goal Modeler.

10.5 Plan Modeler

This role is responsible for designing the agent's plans to achieve a set of goals. This role requires skills to create Finite State Automata and to understand O-MaSE Plan Model specification. This role may interact with the Goal Modeler and Agent Modeler.

10.6 Capability Modeler

This role is responsible for generating the Capability Model. This role requires O-MaSE Capability Model specification knowledge. The Capability Modeler may interact with the Role Modeler, Agent Class Modeler, and Goal Modeler.

10.7 Protocol Modeler

This role is responsible for generating the different protocols that may exist between Agents, Roles, and between Organization and external Actors. This role requires Agent-UML (Odell, Parunak & Bauer 2001) skills and may interact with the Organization Modeler, Agent Class Modeler, and Role Modeler.

10.8 Policy Modeler

This role is responsible for generating the different policies that may govern the Organization. Policies are specified using First Order Predicate Logic, (skill required by this Role), and may constrain Agents, Roles, Capabilities, Plans, etc., therefore this role may interact with the different O-MaSE modelers.

10.9 Action Modeler

This role is responsible for generating the Action Model. This role requires knowledge about the Action Analysis technique. In addition, this role requires knowledge about First Order Predicate Logic notation.